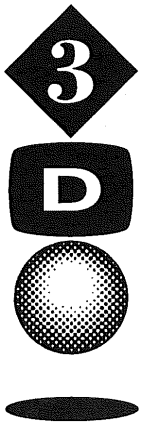


# 3DO M2 Mercury Programmer's Guide

Version 2.7 – June1996

# Contents

# Preface

## About This Book

This manual describes the 3DO M2 Mercury rendering engine and shows how you can use it to create M2 applications.

## About the Audience

This manual is written for title developers. The information presented and the level of description is based on the assumption that you are familiar with both the C programming language, and with three-dimensional graphics.

## System Requirements

To use Mercury, you should have access to a working 3DO M2 development system and to an Apple Power Macintosh or Macintosh Quadra running Operating System 7.5 or higher. You must also have access to a 3DO M2 system installed in accordance with the installation procedures described in the *3DO Development Environment Installation Guide.*

## How This Book Is Organized

Chapter 1, "Introducing Mercury " provides a brief overview of Mercury, describes its components, and lists and describes the steps that are needed to create an M2 application using Mercury.

Chapter 2, "Using Mercury " presents and describes an example program that shows how to develop a simple Mercury application.

Appendix A, "Parts of the Mercury Engine" lists and describes the C-language function calls provided in the Mercury API.

Appendix B, "Matrix Calls" lists and describes Mercury matrix functions.

Appendix C, "Vector Calls" lists and describes Mercury vector functions.

## Typographical Conventions

The following typographical conventions are used in this book:

| Item | Example |
|---|---|
| code example | `Scene_GetStatic(scene)` |
| procedure name | `Char_TotalTransform()` |
| new term or emphasis | In M2, *characters* are objects that can be displayed on the screen. |
| file or folder name | The *remote* folder, the *demo.scr* file. |

## Related Documentation

To understand the material in this manual, and to use Mercury in your application, it is necessary to be familiar with the material in the following 3DO M2 manuals:

- *Getting Started With 3DO M2 Release 2.0*
- The *3DO M2 Debugger Programmer's Guide*
- The *3DO M2 Graphics Programmer's Guide*
- The *3DO M2 Command List Toolkit* manual

In addition, the following books are recommended as valuable sources for information on computer graphics and 3D graphics:

- Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice*: Addison-Wesley, 1990
- Neider, Davis, and Woo. *Open GL Programming Guide*: Addison-Wesley, 1993
- OpenGL Architecture Review Board: *Open GL Reference Manual*: Addison-Wesley, 1993.

*Chapter 1*

# Introducing Mercury

Mercury is a 3D graphics pipeline that is optimized to run in the 4K instruction cache of the Power PC (PPC) 602 CPU that drives M2. Although Mercury is designed to be used by programs written in C or C++, it runs at a speed approaching that of assembly language because all its internal procedures are written in assembly language and pass arguments via registers.

Unlike a conventional C-language API that repeatedly moves data back and forth between CPU registers and a stack, Mercury operates as a simple pipeline that maintains data in a compact data area, paying special attention to data cache alignment.

Because of its efficient and optimized bare-bones design, Mercury can be easily extended and modified to meet the individual needs of each title you develop. Once you understand Mercury's functional interface, you can modify or even replace its functions to suit the requirements of your title and your own personal preferences.

This chapter describes the Mercury rendering engine and explains how to get started using it. To demonstrate basic features of the Mercury system, the chapter uses an example program named *helloworld*, which you can find in the Examples folder on your distribution CD-ROM.

This chapter focuses on the following major topics:



# Understanding Mercury

The heart of the Mercury system is a rendering engine that can be accessed from C or C++ applications. To use the Mercury rendering engine, you provide it with a linked list of objects called pipeline object descriptors (PODs). From this linked list, Mercury produces a command list that is then sent to the M2 Triangle Engine for further processing. (For more information about command lists and the Triangle Engine, see the *3DO Command List Toolkit* manual.)

## Pipeline Object Descriptors (PODs)

When you start creating titles using Mercury, you will become very familiar with pipeline object descriptors because Mercury is a POD-based system. In Mercury, PODs are objects that contain pointers to textures, geometries, transformation matrices, and a wide variety of objects used in rendering, texturing, lighting, and manipulating properties of materials.

Mercury treats each POD in a 3D scene as a geometry entity that is transformed independently. Every POD used in a scene is associated with a texture, geometry, a transform, and a list of the lights that are used to illuminate the POD. To create and implement an application using Mercury, you define a set of PODs that meet your application's requirements and then manipulate them using C-language calls and macros provided by Mercury.

## How PODs Are Used in the helloworld Program

The *helloworld* sample program shows how PODs can be used in a Mercury program. *helloworld* is a simple application that uses just two PODs: one to create and manage a model, and one to create and manage a background. the model POD is named `firstPod`, and the background POD is named `gCornerPod` (because the program displays a fighter aircraft floating in a box that resembles the corner of a room).

The gCornerPod object used in the *helloworld* program is defined as follows in a file named *data.c*:

```
Pod gCornerPod = {
    /* uint32 flags */                          0,
    /* struct Pod *pnext */                     NULL,
    /* void (*pcase)(CloseData*) */             M_SetupPreLit,
    /* struct PodTexture *ptexture */           NULL,
    /* struct PodGeometry *pgeometry */         &gCornerGeometry,
    /* Matrix *pmatrix */                       &gCornerMatrix,
    /* uint32 *plights */                       gCornerLightList,
    /* uint32 *puserdata */                     NULL,
    /* Material *pmaterial */                   &gCornerMaterial
};
```

### Why PODs Are Important

PODs are vital in a Mercury application because almost all operations performed in Mercury depend upon them. In a Mercury-based game, you use PODs to compute bounding boxes, perform lighting operations, and display all backgrounds and game characters. You'll find POD objects referred to repeatedly throughout this chapter and throughout Chapter 2, "Using Mercury ."

For more details about how PODs are constructed and how they are used in Mercury programs, see the section headed "Mercury Geometry" on page 13.

**Note:** *The rendering performance that you can achieve with Mercury varies with factors such as the size of the POD, the number of different textures used in a scene, and both the number of lights used and the kind of lighting that is created. The amount of shared information among PODs, such as textures, also affects rendering performance.*

## Creating an Application with Mercury

Although Mercury can be used to create many different kinds of games, the process of developing an application using Mercury can be broken down into just a few general steps. Those steps can be summarized as follows.

1. Perform the initialization operations your application requires. Initializing a Mercury-based application includes the following steps:
   - Creating and initializing a GState object
   - Initializing Mercury
   - Creating and setting up a camera.

   All three of these operations are covered in more detail in Chapter 2, "Using Mercury ."
2. Perform the work that is needed to create and render each frame in your game. This section of your application contains the code that performs the standard per-frame operations, such as frame-buffering and page-flipping. Code in this section also initializes the M2 destination blender.
3. Write the code that is required to run your game. In this section, you update all POD transforms, and you create and display the specific PODs that are needed at runtime in each stage of your game. Typically, code executed in this section of a game calls various Mercury and GState functions to perform such

operations as displaying rendered images (such as 2D images), displaying 3D Mercury PODs, and handling user input.

4. Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

The sequence of operations described in this list is examined in Chapter 2, "Using Mercury .

# The Mercury API

The Mercury API (application programming interface) is made up of functions that can be called from C-language applications, and data structures that are accessed by those functions. This section briefly describes the functions and data structures provided in the Mercury API. The functions and data structures introduced in this section are covered in more detail under separate headings later in this chapter and in Appendix A, "Parts of the Mercury Engine."

## Mercury's Four Core Functions

Four of the most important function calls provided in the Mercury API are `M_Init`, `M_End`, `M_Sort`, and `M_Draw`. These four functions are vital to the operation of the Mercury system. They perform the following operations:

- `M_Init` initializes the system for rendering. Your application must call `M_Init` before it calls any other Mercury functions.
- `M_End` frees up all memory allocated in `M_Init`, including the memory allocated to the `CloseData` structure passed into this routine.
- `M_Sort` sorts the linked list of PODs that your application provides to Mercury. Mercury must sort the list of PODs that you provide before it can process them.
- `M_Draw` takes your application's list of PODs as input and generates an M2 command list that can then be used for rendering your PODs on the screen.

Mercury's core functions are described in more detail in "The Core Functions" on page 47.

## Other Mercury Functions

Along with the four core functions described under the previous heading, Mercury provides a number of other important functions that perform more specific kinds of operations. These additional functions can be broken down into the following categories:

- *M_Draw . . . functions*, which are called to manage Mercury's drawing operations. The name of each draw function begins with the letters `M_Draw`. For example, the `M_DrawDynLit` function dynamically lights triangles according to the list of lights in each POD. In this book, this group of functions is referred to generically as Mercury's `M_Draw. . .` functions. These `M_Draw. . .` functions should be distinguished from Mercury's main `M_Draw` function (see "Mercury's Four Core Functions," preceding) which is not followed by an ellipsis (. . .) when its name appears in this volume. For more information on Mercury's `Draw. . .` functions, see "M_Draw. . . Functions" on page 50.

- *M_Light. . . functions*, which are used to control lighting in Mercury applications. All of Mercury's light functions have names that begin with the letters `M_Light`. For example, the `M_LightFog` function computes an alpha value that can be used for a fogging effect. In this book, this group of functions is referred to collectively as the `M_Light. . .` functions. For more information on the `M_Light. . .` functions, see "M_Light. . . Functions" on page 51.
- *Matrix functions*, which are used to construct and manipulate matrices. Mercury's matrix functions are listed and described in Appendix B, "Matrix Calls."
- *Vector functions*, which are used to construct and manipulate vectors. the vertex functions are listed and described in Appendix C, "Vector Calls."
- *Utility functions*, which perform operations related to lighting, bounds-testing, cameras, and textures. Utility functions include `M_PreLight`, `M_BoundsTest`, `M_LoadPodTexture`, `M_SetCamera`, `M_Matrix_Perspective`, and others. `M_PreLight` computes lighting values, `M_BoundsTest` tests bounding boxes to determine whether they are visible, and `M_LoadPodTexture` loads textures. `M_SetCamera` takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix (see "Constructing the Skew Matrix" on page 32). The `Matrix_Perspective` function is used to prepare a specially formatted skew matrix that Mercury uses (see "The Matrix_Perspective Function" on page 30).
- *Texture-blending and destination-blending setup functions*, which initialize Mercury's texture blender and destination blender. Each texture-blending function and destination-blending function has a name that begins with the letters `M_TB` (for texture-blending functions) or `M_DB` (for destination-blending functions). For example, the `M_DBInit` function initializes the destination blender state for use with other `M_DB` calls.
- *Case setup functions*: In Mercury, the *case* of a POD is the combination of the state in the Triangle Engine and the draw function used to construct the triangle commands. Each case used in Mercury is equipped with a small *case setup function* that is called when the first POD of the case is encountered—that is, when the `samecaseFLAG` is not set in the POD. When a case setup function is called, it places the draw function used by all PODs in the `CloseData` structure. Then it loads a command list into the Triangle Engine. Mercury provides case functions for 15 predefined cases. These functions are listed in "Case Setup Functions" on page 58, along with a table that shows all legal combinations of case setup functions.

For more detailed descriptions of the functions described in the preceding list, see Appendix A, "Parts of the Mercury Engine."

---

**Note:** *It is important to distinguish between Mercury's* `M_Draw` *function and all other Mercury functions that begin with the letter's* `M_Draw`. *There is only one* `M_Draw` *function, but there are a number of other functions that begin with the letters* `M_Draw`*—for example,* `M_DrawDynLit`, `M_DrawPreLit`, *and* `M_DrawDynLitTex`. *In this book, Mercury's main* `M_Draw` *function is referred to simply as* `M_Draw`, *while the other functions that begin with the letters* `M_Draw` *are referred to collectively as the* `M_Draw. . .` *functions. Also note that Mercury has a set of functions that begin with the letters* `M_Light`. *This group of functions is referred to collectively as Mercury's* `M_Light. . .` *functions.*

---

### Data Structures Used in Mercury

The Mercury engine makes extensive use of a small set of data structures. Many of the function calls used in Mercury work by setting attributes that are defined as fields in these data structures, and many other Mercury calls return the values of the fields in these data structures. The data structures used most often in Mercury applications are:

- The `PodGeometry` structure is a geometry data format used in all of Mercury's draw routines. It is made up of a 44-byte header, a set of values used to define vertices, a set of vertex indices, and a set of texture-map coordinates. For more information on the `PodGeometry` structure, see "The PodGeometry Structure" on page 14.
- The `Pod` structure is the highest-level data structure used in Mercury. The header section of the `PodGeometry` structure, described above, is a `Pod` structure. The job of the `Pod` structure is to associate a POD's geometry with a Matrix, a light list, a surface Material, one or more textures, and the code which Mercury will call to actually render the object. The `Pod structure` is described in more detail in "The Pod Structure" on page 21.
- The `CloseData` structure stores all input to the Mercury pipeline. Mercury also uses the `CloseData` structure as a repository for intermediate calculations and temporary data. For more details about the `CloseData` structure, see "The CloseData Structure" on page 12.
- The `Material` structure contains values that are used to calculate the base, diffuse, shine, and specular colors of each object used in Mercury. The `Material` structure is examined more closely in "The Material Structure" on page 26.

## Mercury Instruction Caching

Mercury is designed to use the Power PC's 4K instruction cache in the most efficient way possible. For example, to ensure that different kinds of routines do not conflict with each other at runtime, Mercury places them in different libraries that can be linked in a specific order when an application is being built. If an application follows this recommended linking sequence at link time, the result is faster execution at runtime.

This section explains how you can take advantage of the caching optimizations built into Mercury's library structure and use those optimizations to design faster-running games.

### Mercury's Library Hierarchy

If you open the Mercury folder provided on your distribution CD-ROM, you'll see that Mercury's code is placed in the folders shown in Table 1-1.

**Table 1-1** *Contents of the Mercury libraries*

| Library | Contents |
|---|---|
| *libmercury1* | Code that implements the `M_Draw` routine |
| *libmercury2* | Mercury's `M_Light...` routines |
| *libmercury3* | Mercury's `M_Draw...` (rendering) routines |

**Table 1-1** *Contents of the Mercury libraries*

| Library | Contents |
|---|---|
| *libmercury1* | Code that implements the `M_Draw` routine |
| *libmercury4* | Miscellaneous routines |
| *libmercury_setup* | Mercury's setup routines |
| *libmercury_util* | Utility routines |

To make sure that the routines in your application cache data and retrieve it as efficiently as possible, it is important to link *libmercury1*, *libmercury2*, and *libmercury3* in the same order in which they are numbered—that is, to link *libmercury1* first, followed by *libmercury2* and then by *libmercury3*.

There is no particular significance in the order in which you link your application with *libmercury4* and *libmercury_util*. That's because *libmercury4* and *libmercury_util* contain only utility routines and other miscellaneous functions.

## How Mercury Instruction Caching Works

Figure 1-1 shows why linking order is important in Mercury-based games. Notice that instructions in Mercury's *libmercury1*, *libmercury2*, and *libmercury3* libraries are cached into different areas of the 4K instruction cache provided in the Power PC.

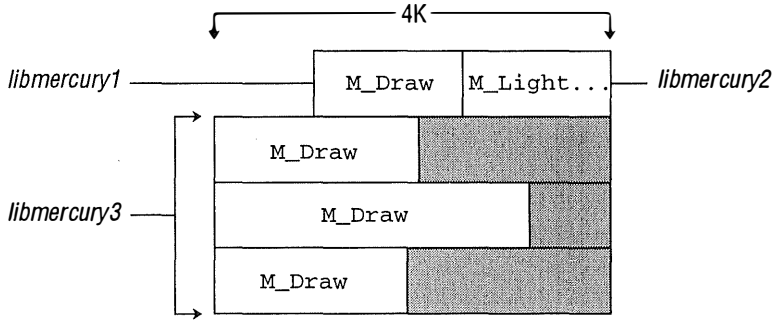


**Figure 1-1** *Mercury Instruction caching*

As you can see, the `M_Draw` and `M_Light...` instructions (implemented in *libmercury1* and *libmercury2*) are cached into memory first, followed by the application's `M_Draw...` functions and their arguments, which are implemented in *libmercury3*.

Because Mercury instructions are cached in this sequence, you can speed up their processing by simply making sure that they are linked in the same order.

## Example: Building a Mercury Application

The order in which you link your game's setup routines is also significant. Most games don't use every setup routine that Mercury provides, but the setup routines that you do use should be linked to your application in the same order shown in Example 1-1 (the makefile used by the *bigcircle* example program).

**Example 1-1** *Linking Mercury Setup Routines*

```
#    @(#) bigcircle.make 96/03/06 1.2

LIBMERCURY = ∂
       -l :::lib:libmercury1:objects:libmercury1 ∂
       -l :::lib:libmercury2:objects:libmercury2 ∂
       -l :::lib:libmercury3:objects:libmercury3 ∂
       -l :::lib:libmercury4:objects:libmercury4 ∂
       -l :::lib:libmercury_utils:objects:libmercury_utils
LIBSETUP = :::lib:libmercury_setup:objects:

OBJECTS = ∂
       :objects:game.c.o ∂
       :objects:gamedata.c.o ∂
       :objects:gamedata2.c.o ∂
       :objects:mainloop.c.o ∂
       :objects:icacheflush.s.o ∂
       ::helloworld:objects:bsdf_read.c.o ∂
       ::helloworld:objects:data.c.o ∂
       ::helloworld:objects:filepod.c.o ∂
       ::helloworld:objects:graphicsenv.c.o ∂
       ::helloworld:objects:scalemodel.c.o ∂
       ::helloworld:objects:tex_read.c.o ∂
       ∂
       {LIBSETUP}M_SetupDynLit.c.o ∂
       {LIBSETUP}M_SetupDynLitFog.c.o ∂
       {LIBSETUP}M_SetupDynLitTex.c.o ∂
       {LIBSETUP}M_SetupDynLitSpecTex.c.o ∂
       {LIBSETUP}M_SetupDynLitFogTex.c.o ∂
       {LIBSETUP}M_SetupPreLit.c.o ∂
       {LIBSETUP}M_SetupPreLitFog.c.o ∂
       {LIBSETUP}M_SetupPreLitTex.c.o ∂
       {LIBSETUP}M_SetupPreLitFogTex.c.o ∂
       {LIBSETUP}M_SetupDynLitTrans.c.o ∂
       {LIBSETUP}M_SetupDynLitTransTex.c.o ∂
       {LIBSETUP}M_SetupDynLitFogTrans.c.o ∂
       {LIBSETUP}M_SetupPreLitTrans.c.o ∂
       {LIBSETUP}M_SetupPreLitTransTex.c.o ∂
       {LIBSETUP}M_SetupPreLitFogTrans.c.o

# Choose one of the following:
DEBUG_OPTIONS = -g # For best source-level debugging
# DEBUG_OPTIONS = -XO -Xunroll=1 -Xtest-at-bottom -Xinline=5

asm = ppcas

# variables for compiler tools
defines=  -D__3DO__ -DOS_3DO=2 -DNUPUPSIM -DMACINTOSH

dccopts=   ∂
             -c -Xstring-align=1 -Ximport -Xstrict-ansi -Xunsigned-char ∂
             {DEBUG_OPTIONS} ∂
             -Xforce-prototypes -Xlint=0x10 -Xtrace-table=0

appname= bigcircle
```

```
LIBS   = -lclt -lspmath -leventbroker -lc ∂
       {LIBMERCURY}

MODULEDIR= "{3doremote}"System.m2:Modules:
BOOTDIR= "{3doremote}"system.m2:Boot:
MODULES= "{MODULEDIR}"graphics "{MODULEDIR}"gstate "{MODULEDIR}"fsutils
"{MODULEDIR}"iff "{BOOTDIR}"kernel "{BOOTDIR}"filesystem
INCLUDEDIR = :::include:

{appname} ƒƒ {appname}.make {OBJECTS}
   link3do -r -D -Htime=now -Hsubsys=1 -Hname={appname} -Htype=5 -Hstack=32768
∂
       -o {appname} ∂
       {OBJECTS} ∂
       -L"{3dolibs}{m2librelease}" ∂
       {MODULES}       #  ∂
       {LIBS}             #  ∂
       > {appname}.map# generate extra map info and redirect from dev:stdout
to a map file
   setfile {appname} -c '3DOD' -t 'PROJ'  # Set creator and type so
                                          # icon appears
   duplicate {appname} -y {3doremote}# copy the goodies to /remote

# files in the object's sub-dir are dependent upon sources in the current dir
:objects: ƒ :

# .c.o files are dependent only upon the .c source
.c.o  ƒ  .c
   dcc -I{INCLUDEDIR} -I::helloworld: {defines} {dccopts} {depDir}{default}.c
-o {targDir}{default}.c.o

.s.oƒ  .s
   {asm} -I {INCLUDEDIR} {depDir}{default}.s -o {targDir}{default}.s.o
```

## Inside Mercury

Because Mercury is designed to function as a pipeline, it performs its internal processing in an order that is predetermined and quite rigid. The sequence in which Mercury performs its processing operations during the execution of an application can be broken down into four stages.

The first stage of processing, called the *sort stage*, is a processing step in which PODs are sorted to optimize the use of TRAM and data caching. To begin the sort stage of processing, you call the `M_Sort` function.

The second, third, and fourth stages of internal processing are called the *POD initialization stage*, the *vertex pipeline stage*, and the *triangle assembly stage*. Mercury carries out all three of these processing stages automatically when you call the `M_Draw` function, which is defined in the *M_Draw.c* file. These three processing stages can be summarized as follows:

- In the *POD initialization stage*, Mercury performs initialization chores required by each POD. In this stage, each of the following operations takes place:
  - Concatenation between the camera matrix (see ***) and the objects matrix
  - Checking the object for screen visibility
  - Initialization of lighting.
- In the *vertex pipeline stage*, the following operations take place:
  - Each vertex in each POD is transformed into screen coordinates
  - If clipping is applied, each vertex is tested for screen visibility
  - Each vertex is lighted (when POD lighting is used).
- The starting point of the triangle assembly stage varies from application to application and from case to case. In the *triangle assembly stage*, the following operations are performed:
  - Vertices are cull-tested
  - Texture coordinates are scaled
  - M2 vertex commands are created and written.

Each of the four stages of Mercury processing is described in more detail under the headings that follow.

## The Sort Stage

When Mercury receives a linked list of PODs from an application, the first thing that happens is that the list is sorted. This stage of processing is called the *sort stage*. To begin this stage of processing, you call the `M_Sort` function, which is implemented in the file *sort.s*.

To sort a POD list, Mercury uses six POD fields: `pcase`, `ptexture`, `pgeometry`, `pmatrix`, `plights`, and `puserdata`.

Using these six fields, Mercury sorts each POD list in ascending order. First, all PODs with the same value in the `pcase` field are sorted according to the address `pcase`. Then all PODs with the same `pcase` values are sorted according to the value in the `ptexture` field, and so on.

To perform its sorting operations, Mercury uses a merge-sort technique that strives to keep all PODs in the data cache.

During this sort stage, Mercury sets the `samecaseFLAG` and `sametextureFLAG` bits in the `flag` field of each POD. Consequently, your application should call Mercury's sort code at least once to make sure that these flags are set.

## The POD Initialization Stage

When you pass a list of PODs to Mercury, a set of initialization operations must be performed on each POD before further processing can begin. The stage of operation in which these initialization tasks are performed is called the *POD initialization stage*.

To begin this stage of processing, an application calls the `M_Draw` function, which is defined in the *M_Draw.s* file. Mercury then starts performing a set of initialization operations that can be broken down into the following substeps:

1. If the *case* of a POD is different from the case of the previous POD—that is, if the POD's `samecaseFLAG` is false—Mercury executes a block of code called *case code* to set up the POD's `CloseData` structure. (The `CloseData` structure is introduced in "The CloseData Structure" on page 12. The *case* of a Mercury POD is the combination of the state in the Triangle Engine and the draw function used to construct the triangle commands. Cases are described in more detail in "Other Mercury Functions" on page 4.)
2. If the texture is different from the texture in the previous POD—that is, if the POD's `sametextureFLAG` is false—Mercury's texture load routine is called to load the PIP and the TRAM with the texture. (For more about PIP tables and texture mapping, see the *3DO Command List Toolkit* manual.)
3. Compute the POD's total transform by concatenating the POD's *pmatrix* transform with the *fcamskewmatrix* transform in `CloseData`.
4. Test the POD's bounding box for visibility on the screen.
5. Invert the POD's total transform.
6. Call the initialization routine for each light source used by a POD.

## The Vertex Pipeline Stage

The *vertex pipeline stage* is the stage of operations in which your application calls Mercury's `M_Draw...` and `M_Light...` utility functions. The `M_Draw...` functions and the `M_Light...` functions are implemented in *.s* files with names that begin with the letters *M_Draw* and *M_Light*, which are in the *libmercury2* and *libmercury3* folders.

Code that is executed during the vertex pipeline stage of processing varies from application to application and from POD case to POD case, but the steps that are used to process the code are always the same during this stage of processing. Specifically each vertex in the POD is:

1. Transformed into screen coordinates
2. Lighted (for lit cases only).

To optimize processing during the vertex pipeline stage, Mercury processes vertices in pairs, two vertices at a time. That means, of course, that an even number of vertices must always exist. If an odd number of vertices is present, your application must create a dummy vertex by simply making a duplicate of the last one.

When vertices are lighted, Mercury's lighting code exits early if both normal vectors in the pair point away from the light. So vertices that have similar normals should be grouped together in pairs.

## The Triangle Assembly Stage

The last stage of Mercury's pipeline processing is the stage in which M2 vertex commands are created and written. This stage of processing is called the *triangle assembly stage*. The code that is executed in this stage varies from case to case. The names of the files that implemented the code also vary from case to case. The entry point associated with this stage is case-specific as well.

Again, however, the steps that are used to process code during this stage do not vary. During the triangle assembly stage, the following operations occur:

1. The texture coordinates are multiplied by *1/w* (for textured cases only).
2. The texture coordinates are scaled in accordance with the size of the texture.
3. Each triangle is tested for back face rejection.
4. The vertices of each triangle are clipped to the screen boundary and clipped in hither.
5. The indices that form strips and fans are converted to M2 strip and fan commands.

## The CloseData Structure

In Mercury, a structure called the `CloseData` structure contains all input to the graphics pipeline. The `CloseData` structure is also used as a repository for intermediate calculations and temporary data.

Many of the function calls used in Mercury access fields that are defined in the `CloseData` structure. Mercury uses several `CloseData` fields in its rendering code, and those fields must be set up by the calling program before the program calls the `M_Draw` function to render its images on the screen.

Table 1-2 lists the fields of the `CloseData` structure that must be filled in before your application calls `M_Draw`.

**Table 1-2** *CloseData Fields that Your Application Must Supply*

| Field | Contents |
|---|---|
| `float fwclose` | *w* value for the hither plane |
| `float fwfar` | *w* value for yon plane |
| `float fscreenwidth` | screen width in pixels |
| `float fscreenheight` | screen height in pixels |
| `uint32 fogcolor` | the packed fog color (output by `M_PackColor`) only needed if using fog |
| `uint32 srcaddr` | address of the pixels in the current frame buffer only needed if using transparency |
| `uint32 depth` | depth, in bits, of the current frame buffer, only needed if using transparency |
| `float fcamx, fcamy, fcamz` | position of the camera in world coordinates |
| `Matrix fcamskewmatrix` | a matrix that transforms world coordinates into screen coordinates |

## Mercury Geometry

Mercury 3D geometry can be defined in terms of independent rigid bodies, each possessing a transformation that orients and places it in world coordinates. The observer is moved around by a mechanism called the *camera transformation*, which maps the world coordinate system into the screen coordinate system.

Even though the transformations of individual pieces used in a game may be hierachically related, the vertices of the geometry used to render those pieces must ultimately be transformed into the coordinate system used by M2. Consequently, Mercury splits up the CPU cycles of the 602 into the following tasks:

- Game play
- Transforming and lighting vertices
- Assembling the triangles formed between the vertices into M2 commands.

**Note:** *In the preceding list, the term game play includes such tasks as updating the transformations of individual objects in the scene and updating the camera every frame.*

### Geometry Structures Used in Mercury

Mercury uses a geometry model called the `PodGeometry` data structure in all its draw routines. The `PodGeometry` structure provides all of Mercury's components with a consistent structure for representing geometry objects, and the result of this consistency is a combination of high performance and flexibility.

Encapsulating geometry objects into `PodGeometry` structures has numerous advantages. For example, when a model is to be rendered, it does not matter whether the model is textured or untextured, whether it is pre-lit or dynamically lit, or whether it is transparent or opaque. The underlying data structure used to render the model is the same.

Other advantages of using `PodGeometry` structures are:

- By calling a function named `M_Prelight`, you can convert models converted from dynamically lit models (with normals) to pre-lit models (with colors) .
- You can easily change the texture used to render a model because Mercury uses normalized $u$ and $v$ coordinates.
- Vertices and colors can be shared to create faceted objects.

## The PodGeometry Structure

A `PodGeometry` structure is a C-language `struct` that is defined as follows:

```
typedef struct PodGeometry
{
     float fxmin,fymin,fzmin;
     float fxextent,fyextent,fzextent;
     float *pvertex;
     short *pshared;
     uint16 vertexcount;
     uint16 sharedcount;
     short *pindex;
     float *puv;
} PodGeometry;
```

The following `PodGeometry` structure is used in the example *helloworld* program:

```
static PodGeometry gCornerGeometry = {
     /* float fxmin, fymin, fzmin */ -100.0, -100.0, -100.0,
     /* float fxextent, fyextent, fzextent */ 200.0, 200.0,
                                              200.0,
     /* float *pvertex */     gCornerVertices,
     /* short *pshared */     gCornerSharedVerts,
     /* uint16 vertexcount */ sizeof(gCornerVertices)/
                              kSizePerVertex,
     /* uint16 sharedcount */ sizeof(gCornerSharedVerts)/
                              (2 * sizeof(short)),
     /* short *pindex */      gCornerIndices,
     /* float *puv */         NULL
};
```

## Example: Using Mercury Geometry to Construct an Object

Example 1-2 shows how the `PodGeometry` structure can be used in a Mercury application. The example constructs a faceted cube using Mercury geometry. It demonstrates the use of vertices, indices, $(u, v)$ coordinates, and a `PodGeometry` header.

**Example 1-2** *Using Mercury Geometry*

```
static float facetcubevtx[] =
{
   0,0,0, -1,0,0,       /* 0-x */
   0,0,100,0,-1,0,      /* 1-y */
   0,100,0,0,0,-1,      /* 2-z */
   0,100,100,0,1,0,     /* 3+y */
   100,0,0,1,0,0,       /* 4+x */
   100,0,100,0,0,1,     /* 5+z */
   100,100,0,1,0,0,     /* 6+x */
   100,100,100,1,0,0,   /* 7+x */
};

static short facetcubeshared[] =
```

```
{
    0,1,        /* 8-y */
    0,2,        /* 9-z */
    1,0,        /* 10-x */
    1,5,        /* 11+z */
    2,0,        /* 12-x */
    2,3,        /* 13+y */
    3,0,        /* 14-x */
    3,5,        /* 15+z */
    4,1,        /* 16-y */
    4,2,        /* 17-z */
    5,4,        /* 18+x */
    5,1,        /* 19-y */
    6,3,        /* 20+y */
    6,2,        /* 21-z */
    7,3,        /* 22+y */
    7,5,        /* 23+z */
};

static short facetcubeindex[] =
{
    0 + STXT + PCLK,2,10,12,14,      /* -x */
    8 + NEWS + CFAN + PCLK,16,1,19, /* -y */
    9 + NEWS + CFAN + PCLK,2,17,21, /* -z */
    13 + NEWS + CFAN + PCLK,3,20,22,/* +y */
    6 + NEWS + CFAN + PCLK,7,4,18,  /* +x */
    11 + NEWS + CFAN + PCLK,5,15,23,/* +z */
    0 + STXT + NEWS + CFAN + PCLK,-1,
};

static float facetcubeuv[] =
{
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
    0,0,0,1,1,0,1,1,
};

PodGeometry facetcube =
{
    0,0,0,
    100,100,100,
    facetcubevtx,
    facetcubeshared,
    sizeof(facetcubevtx)/(6*sizeof(float)),
    sizeof(facetcubeshared)/(2*sizeof(short)),
    facetcubeindex,
    facetcubeuv
};
```

### Parts of a PodGeometry Structure

A `PodGeometry` object is divided into five parts:

- A 44-byte header
- A set of vertices
- A set of shared vertices
- Vertex indices for the object's strip fans (one `short` integer per index)
- The corresponding texture-map ($u, v$) coordinates (two `float` values for each $u, v$ pair)

Each of these four parts of a `PodGeometry` object is described individually under the headings that follow.

### The PodGeometry Header

The fields in a `PodGeometry` header are defined as follows:

| Field | Contents |
|---|---|
| `xmin, ymin, zmin` | The minimum $x, y$, and $z$ coordinates used in the object. |
| `xextent, yextent, zextent` | The maximum $x, y$, and $z$ values used in the object, minus the minimum values used. |
| `pvertex` | Points to the array of vertices used. A vertex is an array of floats containing the coordinates of the vertex and either a normal vector (for lit cases) or a color (for pre-lit cases). |
| `pshared` | Points to an array of shared vertices |
| `vertexcount` | The number of vertices in `pvertex` |
| `sharedcount` | The number of shared vertices in `pvertex` |
| `pindex` | The array of indices. |
| `puv` | The array of ($u, v$) coordinates there is one ($u, v$) coordinate for each index in pindex. |

### The PodGeometry Vertices and Shared Vertices

The list of vertices in a `PodGeometry` structure is divided into two parts: a list of the actual vertices, followed by a list of `short` integers that point to the vertex and color information that is to be copied. In a `PodGeometry` structure, a vertex is made up of six floating-point numbers: specifically, $x, y$, and $z$ followed by either $nx, ny$, and $nz$ or $r, g$, and $b$. A shared vertex is made up of two short integers: a vertex index followed by a color index.

This architecture permits rapid execution when used with faceted geometry. As mentioned previously, Mercury vertices are transformed and lighted two at a time, so a dummy vertex must be added if there is an odd number of vertices.

Shared vertices are copied one at a time, so there is no need to pad your vertices out to an even number.

Faceted geometry uses only one normal for an entire face of an object. Consequently, color calculations can be reduced by calculating the color for only one of the points being used, and then copying that color along with vertex information for subsequent instances of the same normal. For example, if your application displays a faceted cube that has eight vertices and six normals, you

would normally have to transform and light 24 vertices to display the cube. By using Mercury's shared mechanism, you can light only eight vertices, and then make 16 copies. Of course, this solution yields correct lighting only with directional light, but in most cases it really doesn't look bad with the other kinds of lighting; try it and see.

#### The PodGeometry Indices

Two kinds of half-words used to describe pod geometry indices: vertex indices and texture numbers. Should a new texture need to be selected for a strip fan (which is always the case with the first strip fan in the object), the first vertex index has a magic bit set, and is followed by the number of the texture to select (a small integer). Requesting a load of texture `-1` terminates the object.

There are five bits per index, so this format supports up to 2,048 vertices in a model:

```
#define PCLK0x8000
#define PFAN0x4000
#define CFAN0x2000
#define NEWS0x1000
#define STXT0x0800
```

Later in this section, each of the flags shown in this code fragment is described under a separate heading. Before we examine each individual flag, however, it might be helpful to take a look at how triangles are drawn in Mercury-based programs. That's because many of the flags used in the PodGeometry structures are used to determine how triangles are drawn in strips and fans. When you understand how triangles are drawn in a Mercury-based application, you'll have a better understanding of the flags that are provided in Mercury's PodGeometry structure.

The next subsection, "Drawing Clockwise and Counter-Clockwise Triangles," explains how triangles are drawn in Mercury programs.

### Drawing Clockwise and Counter-Clockwise Triangles

Mercury uses a very fast set of algorithms for drawing triangles in strips and fans. To take advantage of this optimization when you construct a model, it helps to know whether the triangles used to render the model are drawn clockwise or counter-clockwise. Making this determination involves one small trade-off: the techniques used for drawing clockwise and counter-clockwise triangles in Mercury are not quite as intuitive as some older, more traditional strategies that you may be familiar with. But once you understand the Mercury system, it is not very difficult to use, and the extra speed is worth the extra effort.

Briefly, here's how the system works: When you want to draw a triangle in a strip or a fan, you use a flag called `prevclockwiseFLAG` to determine whether the previous triangle was clockwise or counter-clockwise.

#### Drawing Triangles in a Strip

If you are drawing triangles in a strip, Figure 1-2 shows what happens when you draw your triangles in this fashion. In the strip shown in Figure 1-2, the triangle at the left—which we will call Triangle 1 because it appears beneath the number 1—is the triangle that is drawn first. Notice that Triangle 1 is drawn in a clockwise

direction. The second triangle—which we will call Triangle 2 because it appears above the number 2—is drawn counter-clockwise. The third triangle, or Triangle 3, is drawn clockwise, and so on.

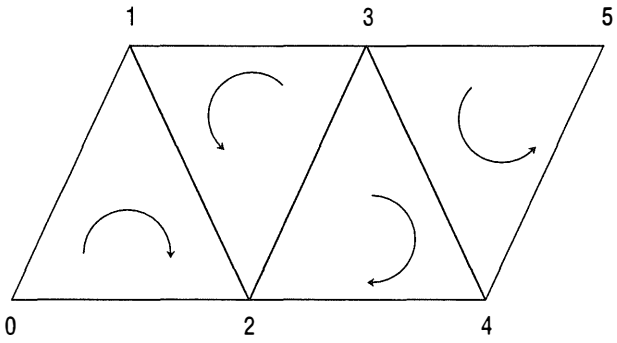


**Figure 1-2** *Drawing triangles in a strip.*

When you are drawing triangles in a strip, another rule to remember is that each time you add a triangle, you replace the oldest vertex in the strip. This process continues until a complete strip is drawn. (When you draw triangles in a fan configuration rather than in a strip, the process is slightly different, as you'll see momentarily.)

Table 1-3 shows more details about how triangles are drawn in a strip using the "replace-oldest-vertex" rule.

**Table 1-3** *Using the "Replace-Oldest-Vertex" Rule*

| Slot | Triangle 1 | Triangle 2 | Triangle 3 | Triangle 4 |
|---|---|---|---|---|
| **A** | 0 | **3** | 3 | 3 |
| **B** | 1 | 1 | **4** | 4 |
| **C** | 2 | 2 | 2 | **5** |
| **CW or CCW[a]** | CW | CCW | CW | CCW |

a. Clockwise or counter-clockwise

Table 1-3 is just another mechanism for showing how the triangles illustrated in Figure 1-2 are drawn. Notice that the left–most column of Table 1-3 is labeled *Slot*. In Mercury, the word *slot* is used to describe a kind of container in which a number can be placed—in much the same way that a value can be placed in a register in the world of computer hardware.

When you are determining whether rotation is clockwise or counter-clockwise, follow the path from the vertex in Slot A to the vertex in Slot B to the vertex in Slot C.

For example, Table 1-3 shows how a group of three slots—named A, B, and C—can be used to hold a set of numbers that represent the three vertices of a triangle. This is a convenient mechanism for showing how the vertices of new triangles replace vertices of older triangles as new triangles are added to a strip. When you

draw a triangle, you can specify the order in which its vertices are drawn by a placing the vertex number into the appropriate slot. Then, each time a vertex of a new triangle replaces a vertex of an older triangle, you can specify which vertex of the older triangle is replaced by simply removing the old vertex from its slot and replacing it with the new one.

For example, refer to Triangle 1 in Table 1-3. When that triangle is drawn, Vertex 0 is placed in Slot A, Vertex 1 is placed in Slot B, and Vertex 2 is placed in Slot C.

Whether a triangle is drawn clockwise or counter-clockwise is determined by looking at the direction (clockwise or counter-clockwise) of the three vertices in Slot A, B, and C, in that order.

When it is time to draw Triangle 2, it is drawn in a counter-clockwise direction. Its vertices are numbered 1, 2, and 3, and Vertex 3 in Triangle 2 replaces Vertex 0 in Triangle 1, as shown in the table. This process continues as you add more triangles to the strip shown in Figure 1-2. When you draw Triangle 3, Vertex 4 replaces Vertex 1. When you draw Triangle 4, Vertex 5 replaces Vertex 2—and so on.

### Drawing Triangles in a Fan

There is only one difference between drawing a triangle fan in Mercury and drawing a triangle strip. (Whether group of triangles is drawn in a strip or in a fan is determined by the `pfan` and `cfan` flags, as you'll see later in this section.) When you add a triangle to a strip, you don't replace the oldest vertex, as you do when you add a triangle to a strip. Instead, you replace the middle vertex of the most recently drawn triangles. This difference is illustrated in Figure 1-3.

Table 1-4 on Page 20 shows how the "replace-middle-vertex" rule works when you add a triangle to a fan.

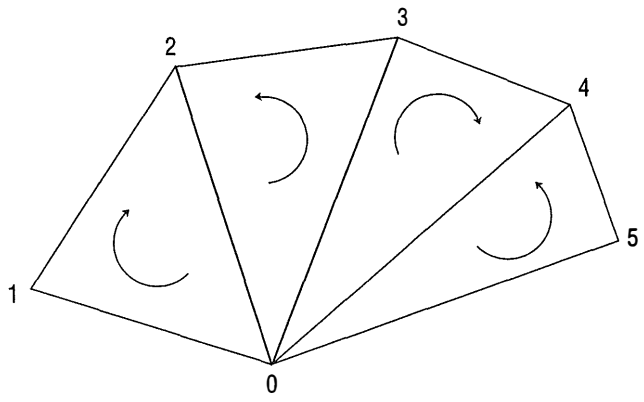


**Figure 1-3** *Adding triangles to a fan.*

**Table 1-4** *Using the "Replace-Middle-Vertex" Rule*

| Slot | Triangle 1 | Triangle 2 | Triangle 3 | Triangle 4 |
|---|---|---|---|---|
| **A** | 0 | 0 | 0 | 0 |
| **B** | 1 | **3** | 3 | **5** |
| **C** | 2 | 2 | **4** | 4 |
| **CW or CCW[a]** | CW | CCW | CW | CCW |

a. Clockwise or counter-clockwise

To see what happens when you add a triangle to a fan, refer to Triangle 1 in Table 1-4. That triangle is drawn in the same way it was drawn earlier, when we added a triangle to a strip. Vertex 0 is placed in Slot A, Vertex 1 is placed in Slot B, and Vertex 2 is placed in Slot C.

When it is time to draw Triangle 2, however, the rules change. This time, Vertex 3 replaces Vertex 1 in Triangle 1. When you draw Triangle 3, Vertex 4 replaces Vertex 2 in Triangle 2. When you draw Triangle 4, Vertex 5 replaces Vertex 3—and so on.

## Flags Used in the PodGeometry Structure

Now that you know how triangles are draw in Mercury applications, it's time to examine the flags that are provided in Mercury's PodGeometry structure. In the paragraphs that follow, each flag used in the PodGeometry structure is described under a separate heading.

### The PCLK Flag

The `prevclockwiseFLAG` is used if the previous index had a clockwise orientation. In both strips and fans, the orientation of triangles alternates between clockwise and counter clockwise, as shown in Figure 1-2 and Figure 1-3.

### The PFAN and CFAN Flags

You should use these flags if the previous or current index has a "fan" type orientation (which means that the middle point of the previous triangle should be eliminated.)

---

**Note:** *Because of the way in which the branching order works in the Mercury pipeline, the curfanFLAG must be set whenever the startnewstripFLAG is set.*

---

### The NEWS Flag

Use this flag if your index is the start of a new strip fan. Use of this on the first index in the object is optional.

### The STXT Flag

You can use this flag if a different texture is needed for a strip fan. The next half-word is the texture number. The first strip fan in an object always needs a texture.

### The PodGeometry (u, v) Coordinates

In a PodGeometry object, the $(u, v)$ coordinates are normalized floating-point values. The value 0.0 is the smallest texture coordinate, and the valuie 1.0 is the largest. The normalized coordinate is scaled at run time based on the size of the biggest LOD of the texture. When a texture within a texture page is selected the draw routine also loads the *uscale* and *vscale* of the texture. If tiling is being used, the coordinate can be greater than 1.0.

## The Pod Structure

The `Pod` structure is the highest-level data structure in Mercury. It associates a POD's geometry with a `Matrix`, a light list, a surface `Material`, one or more textures, as well as with the code which Mercury will call to actually render the object.

Benefits that Mercury derives from using `Pod` structures include the following:

- The user can easily change how an object looks by choosing different draw routines, textures, material properties, and lighting.
- A pointer to user data is also provided in the Pod structure, to allow custom lighting or rendering code to access user-specifiable information.
- The `Pod` structure contains a pointer to the next POD, permitting PODs to be grouped into linked lists.

The following `Pod` structure is used in the *helloworld* sample program:

```
Pod gCornerPod = {
     /* uint32 flags */                 0,
     /* struct Pod *pnext */            NULL,
     /* void (*pcase)(CloseData*) */    M_SetupPreLit,
     /* struct PodTexture *ptexture */ NULL,
     /* struct PodGeometry *pgeometry */&gCornerGeometry,
     /* Matrix *pmatrix */              &gCornerMatrix,
     /* uint32 *plights */              gCornerLightList,
     /* uint32 *puserdata */           NULL,
     /* Material *pmaterial */         &gCornerMaterial
};
```

### Declaration of the Pod Structure

The declaration of the Pod structure is defined as follows:

```
typedef struct Pod
{
     uint32 flags;
     struct Pod *pnext;
     void (*pcase)(CloseData*);
     struct PodTexture *ptexture;
     struct PodGeometry *pgeometry;
     Matrix *pmatrix;
     uint32 *plights;
     uint32 *puserdata;
     Material *pmaterial;
}
```

### Flags Used in the Pod Structure

In most cases, the Mercury user sets the `flags` field of a `Pod` structure to 0. Then, when the `M_Sort` function is called, it optimizes the rendering of the objects by setting other flags. The `flags` field in the Pod structure is currently used for the following attributes:

```
#define samecaseFLAG(1 << (31-8))      /* the 0x00800000 bit */
#definesametextureFLAG(1 << (31-9))    /* the 0x00400000 bit */
#define callatstartFLAG(1 << (31-16))  /* the 0x00008000 bit */
#define casecodeisasmFLAG (1 << (31-17)) /* the 0x00004000 bit */
#define usercheckedclipFLAG(1 << (31-18))/* the 0x00002000 bit */
#define specularFLAG(1 << (31-22))     /* the 0x00000200 bit */
#define hithernocullFLAG(1 << (31-23)) /* the 0x00000100 bit */
#define nocullFLAG(1 << (31-28))       /* the 0x00000008 bit */
#define frontcullFLAG(1 << (31-29))    /* the 0x00000004 bit */
#define clipFLAG(1 << (31-30))         /* the 0x00000002 bit */
#define callatendFLAG(1 << (31-31))    /* the 0x00000001 bit */
```

Each flag is described individually in the following paragraphs.

#### samecaseFLAG

Set this flag if a POD calls the same draw routine (using the `pod->pcase` field) as the previous POD. If this flag is set, the initialization routine for the draw routine does not have to be called (because it was called for the previous POD). The `M_Sort` function sets this flag automatically.

#### sametextureFLAG

Set this flag if a POD uses the same texture page as the previous POD. `M_Sort` sets this flag automatically, to minimize the number of texture loads at render time.

#### callatstartFLAG

If this flag is set, the routine pointed to by `puserdata` executes before calling the `M_Draw...` function.

#### casecodeisasmFLAG

If this flag is set, the `M_Draw` function does not save its registers off before calling `pod->pcase`. If this flag is not set, integer and floating-point registers are saved in RAM, and the `pcase` routine is called using C calling conventions (in particular, GPR3 contains a pointer to `CloseData`.)

#### usercheckedclipFLAG

If this flag is set, `M_Draw` does not check the object to make sure it is within the view volume. The POD being processed should be checked by user code to ensure that it does not clip with the left or top edges of the screen or the hither plane. Otherwise, the results are unpredictable.

#### specularFLAG

If this flag is set the camera position is inverse transformed into local object coordinates. This flag must be set for specular lighting to work properly.

#### hithernocullFLAG

If this flag is set and the object's bounding box is projecting through the hither plane, culling is disabled. Result: The Triangle Engine draws all triangles, whether back-facing or front-facing.

#### nocullFLAG

This flag disables backface culling for the affected POD. Result: The Triangle Engine draws all triangles, whether back-facing or front-facing.

#### frontcullFLAG

Normally, backface culling is enabled. If this flag is set, frontface culling is enabled instead.

#### clipFLAG

This flag is set by the bounding-box check in `M_Draw`. It signals to the rendering routine that clip-testing is enabled. If the `usercheckedclipFLAG` is set, this flag must also be set by the user.

#### callatendFLAG

If this flag is set, the routine pointed to by `puserdata` runs after calling the `M_Draw...` function.

#### Next-Pod Pointer (the pod->pnext field)

The `pod->pnext` field typically points to a "next pod to be rendered." `M_Sort` quickly arranges the linked list of PODs by changing only this field. Note that when calling `M_Sort`, a pointer to NULL does not signify an end-of-list situation. Rather, the user must specify a list length when calling `M_Sort`.

#### Setup Case (the pod->pcase field)

The `pod->pcase` field is a pointer to the function to be called to render the affected object. `M_Draw` calls this routine for the first object that needs this case (see the `samecaseFLAG`, described previously). Typically, this routine is a setup function such as `M_SetupPreLit` or `M_SetupDynLitTex`.

#### Pod Texture (the pod->ptexture field)

The `pod->ptexture` field is a pointer to the `PodTexture` structure describing the texture to be used for the affected POD. For untextured objects, this field is ignored.

#### Pod Geometry (the pod->pgeometry field)

The `pod->pgeometry` field is a pointer to a `PodGeometry` structure. This structure contains information about the vertices, texture coordinates, normals, and the alike.

#### Matrix (the pod->pmatrix field)

The `pod->pmatrix` field points to a `Matrix` data structure. A `Matrix` structure is a structure containing the local transform for a given object. A large matrix library is provided to perform operations for orientating and positioning an object by manipulating the affected matrix.

#### Light List (the pod->plights field)

The `pod->plights` field points to a list of lights to be applied to the object. The list, which is actually just an array of unsigned 32-bit integers, can be shared between multiple PODs that need similar lighting applied to them. The light list is described in the section where `Lighting` data structures are described.

#### User Data (the pod->puserdata field)

The `pod->puserdata` field points to an assembler function that is called if either the `callatstartFLAG` or `callatendFLAG` is set.

#### Material (the pod->pmaterial field)

The `pod->pmaterial` field points to a Material data structure. (The `Material` data structure is described in "The Material Structure" on page 26.)

## Mercury Lighting

Mercury's lighting functions are multi-purpose procedures; along with performing lighting operations, Mercury's light functions can also manage specific per-vertex functions, such as creating fog.

The lighting used with a POD must match the requirements of the Draw function that is used with the pod, and must also match the texture blender and destination blender initialization operations used for the POD's case.

### Mercury's Light Functions

The light functions available in Mercury are:

- `M_LightFog`
- `M_LightDir`
- `M_LightPoint`
- `M_LightSoftSpot`
- `M_LightFogTrans`
- `M_LightDirSpec`
- `M_LightDirSpecTex`

Mercury's light functions work a little differently from the other kinds of functions implemented in the Mercury API. In a Mercury-based game, you don't call Mercury's lighting routines directly. Instead, you construct a list of lights that are visible to each object or group of objects you want to illuminate. In this list, you associate each lighting function that you want to call with the POD object that you want to illuminate. Then, when you call the `M_Draw` function to draw each object that is to be illuminated, Mercury automatically provides the kind of lighting that you have specified.

Along with constructing a light list, you supply Mercury with two other kinds of data structures for each POD you want to light: a *lighting structure* that describes the attributes of the light you want to create for the POD, and a *Material structure* that describes the kind of material that is being illuminated. When you have provided these two structures, Mercury has all it needs to call the appropriate lighting functions.

## Light Lists

As noted in the previous section, every application that uses Mercury's lighting functions is equipped with a *light list*. A light list is a data structure that lists the lighting functions used in your application and some of their attributes. For example, in the *helloworld* example program, the following light list is defined in a file named *filepod.c*:

```
uint32 gModelLightList[] = {
     (uint32)&M_LightDir,
     (uint32)&gDir,
     (uint32)&M_LightPoint,
     (uint32)&gPoint,
     0
};
```

In Mercury, a light list is a list of pointers to light functions and data structures that are used by those functions. Every light function used in Mercury is associated with a specific kind of data structure that is used to define the characteristics of the lighting that the function provides. In a light list, each function that is listed is followed by a pointer to its corresponding data structure. The data structures used with Mercury's light functions are described in more detail under the next heading, "Lighting Structures and Material Structures."

When you have provided your application with a light list, your light list determines what kinds of operations take place when your application calls the `M_Draw` command.

## Lighting Structures and Material Structures

Recall that each light function used in Mercury is associated with two data structures: a *lighting structure*, which describes the kind of lighting you want to use to illuminate a particular POD, and a *Material structure*, which sets the material properties of the POD that is being illuminated. Each lighting function defined in Mercruy uses a different kind of lighting structure. But Mercury defines only one kind of Material structure, which is used by all lighting functions.

Lighting structures are described in more detail under the next heading, "How Lighting Structures Work." Material structures are examined in the subsection headed "The Material Structure" on page 26.

### How Lighting Structures Work

Because different kinds operations have different effects, each light function used in Mercury is associated with a different kind of data structure that must be filled in before its corresponding light function is called. For example, before the *helloworld* program calls the `M_LightDir` function, the program sets the fields of the data structure `LightDir` to the values shown in the following code fragment.

When you call the lighting function that uses the data structure you have filled in, the function computes the *r*, *g*, *b*, and *a* components (or as many of those components as needed) for the surface at the point you have specified.

The *helloworld* program uses two lighting structures—a `LightPoint` structure named `gPoint` and a `LightDir` structure named `gDir`—which are defined in the *filepod.c* file:

```
static LightPoint    gPoint = {
     /* float x, y, z */       -60.0, 60.0, 60.0,
     /* float maxdist */       25000.0 * 256.0 * 0.9,
     /* float intensity */     25000.0,
     /* Color3 lightColor */ { 0.7, 0.7, 0.9   }
};

static LightDir      gDir = {
     /* float nx, ny, nz */     0.3, -0.7, 0.648,
     /* Color3 lightcolor */     0.2, 0.2, 0.2
};
```

Pointers to structures `gPoint` and `gDir` appear in the light list presented earlier in the section, under the heading "Light Lists" on page 25.

The *helloworld* program associates each of these structures with a POD in the following , **"**, which appears in the *filepod.c* source file:

```
*maxPodVerts = 0;
curPod = gBSDF->pods;

for (i=0; i<gBSDF->numPods; i++) {
   if (curPod->pgeometry->vertexcount > *maxPodVerts)
        *maxPodVerts = curPod->pgeometry->vertexcount;
   Model_Scale(curPod, scaleFactor);
   curPod->plights = gModelLightList;
   curPod = curPod->pnext;
}
```

Once this code is executed, the specified kind of lighting is created each time `M_Draw` is called to draw one of PODs shown in the preceding example.

### The Material Structure

Along with a lighting structure that defines the kind of lighting being created, each Mercury lighting function is also associated with a Material structure that defines the characteristics of the object being lighted.

Each POD object used in Mercury is equipped with a pointer named `pmaterial`, which points to a Material structure. In Mercury, a Material structure is defined as follows:

```
typedef struct Material
{
   Color4 base;
   Color3 diffuse;
   float shine;
   Color3 specular;
} Material
```

Mercury's `Material` structure is a little different from materials structures used in some other APIs. The main difference is that in Mercury's `Material struct`, the ambient and emissive values have been combined into the base color. The formula to combine the colors is:

*Ma* * *La* + *Me*

where:

- *Ma* is the ambient material property
- *La* is the ambient light in the scene
- *Me* is the emissive material property.

Since this only needs to be calculated when the ambient light in the scene changes, Mercury combines them to reduce the number of per-object calculations.

If the object is transparent, the user puts the alpha base value in the `base` color.

The `diffuse` color is multiplied by the value of each light, and a `diffuse` color for each light used is stored in a buffer named `convlightdata`.

The `shine` value is the specular exponent for the object. It defines how wide the highlight is.

The `specular` color is multiplied by the value of each specular light, and a specular color for each light used is stored in the `convlightdata` buffer for each specular light used.

For each object being lighted, a `base` color and a `diffuse` color are copied into `closedata`. If the `specularFLAG` is set the shine and specular color are also copied into `closedata`. In addition, the camera's position is inverse-transformed into the object's local coordinates.

## Using Lighting Structures and Material Structures

To specify a lighting function and the pointer to its lighting structure, set the `plights` field in the POD structure. Similarly, set the `pmaterial` field in the POD to specify the `Material` structure.

The `plights` field points to a zero-terminated list of pointer tuples. The first member of the tuple is an address of the light routine. The second tuple member is a pointer to the data that represents the light. To call a lighting function, first initialize the data that the call requires, and then make the call.

## The Calling Sequence of Lighting Operations

One important characteristic of Mercury's lighting routines is that they are always called in a specific sequence. The first time you call a Mercury lighting routine, its input is the *rgb* code of the base color, and its *u* value is 1.0. When the first lighting routine returns, its output—which is returned in the form *r*, *g*, *b*, *a*—is used as the input for the next lighting routine that is called. This process continues until all lighting routines have been called.

More details about the parameters, return values, and other attributes used with Mercury's light functions see Appendix A, "Parts of the Mercury Engine."

## Varieties of Lighting Used in Mercury

In addition to the object's color, the lighting code in Mercury can compute three different kinds of lighting effects: fog, transparency with fog, and specular lighting.

### Fog Lighting

Mercury's lighting code computes an alpha value that is passed unchanged through the texture blender. Then the alpha value goes to the destination blender, where it is blended with the fog color in a constant register using the following formula:

*(alpha* texture blender color) + (fog color * (1-alpha))*

### Transparency with Fog

The lighting code also computes a `UCoordColor` value that is used to look up an intensity value from a special texture map that is 17 entries long. `UCoordColor=0` maps to 0, and `UCoordColor=16` maps to `0xff`. The texture intensity is blended in the texture blender with the fog color in a constant register using the formula:

*(UCoordColor* color) + (fog color * (1-UCoordColor))*

The final color is then destination blended with the frame buffer using the vertex alpha value for the transparency effect.

### Specular Lighting

To create specular lighting, the alpha value computed by Mercury's lighting code is passed unchanged through the texture blender. Then it also goes to the destination blender, where it is multiplied by the specular color in a constant register and added to the output of the texture blender color using the following formula:

*texture blender color + (alpha*specular color)*

Appendix A, "Parts of the Mercury Engine," explains how each of these varieties of lighting works when you call each of Mercury's lighting functions.

---

**Note:** *Mercury's lighting code cannot compute transparency with texture and specular lighting in a single pass because that kind of operation requires the use of the destination blender for two different operations: specular and transparency.*

---

## The Two Stages of a Mercury Lighting Routine

Mercury executes its lighting routines in two distinct stages: an *initialization* stage and a *per vertex* stage.

### The Initialization Stage

In the initialization stage of lighting operations, your application calls routines to initialize all the parameters that are required by each lighting function in the light list. These operations include multiplying the diffuse and specular colors by the value of the light color, and then transforming the light's direction and position into the POD's local coordinates.

### The Per Vertex Stage

In the per vertex stage, the colors for the vertex are calculated. If fog is used, it is also included in the light routines executed at this stage of processing. If fog is calculated, the result of the calculation is returned either in the alpha channel or (in the case of transparency with fog) in the *u* texture coordinate.

Appendix A, "Parts of the Mercury Engine," explains how the initialization and per-vertex stages work in each of Mercury's lighting routines.

## Example: Using Lighting Data

The following code fragment is an example of light data that might be used to represent a POD that has fog, one directional light source, and one point light source.

**Example 1-3** *Lighting Data*

```
LightFog LightFogData =
{
   1.0/(30.0 - 10.0),10.0/(30.0 - 10.0)
};

LightDir LightDirData =
{
   -0.866,0.0,-0.5,
   0.25,0.25,0.25,
};

LightPoint LightPointData =
{
   -0.866,0.0,-0.5,
   250000.0 * 32.0 * 1.0,
   250000.0,
   1.0,1.0,1.0,
};

long planelights[] =
{
   (long)&M_LightFog,
   (long)&LightFogData,
   (long)&M_LightDir,
   (long)&LightDirData,
   (long)&M_LightPoint,
   (long)&LightPointData,
   0
};
```

# Camera Operations in Mercury

One of the first steps in developing a Mercury-based application is to create and set up a *camera skew matrix*. A camera skew matrix is a specially formatted matrix that is stored in Mercury's `CloseData` structure and is used to transform vertices into screen coordinates.

To construct the camera skew matrix for Mercury, two special base matrices are composited together using the equation

$F = CS$

where $F$ is a camera skew matrix, $C$ is a camera coordinate transformation, and $S$ is a source skew matrix .

The camera coordinate transform (C) transforms the world coordinate system in such a way that the objects in the scene are down the negative z axis from the origin. The skew matrix ($S$) scales into the screen coordinates and provides the desired perspective.

To simplify the process of creating and maintaining the camera skew matrix, Mercury provides a pair of utility functions named `M_SetCamera` and `Matrix_Perspective`. These two functions are described in more detail under the headings that follow.

For an example that shows how a camera skew matrix can be set up in a Mercury program, see the subsection under the heading "Creating and Setting Up a Camera" on page 41.

## The M_SetCamera Function

`M_SetCamera` takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix.

The syntax of the `M_SetCamera` function is:

```
void M_SetCamera( CloseData *close, Matrix *normalCameraMatrix,
    Matrix *specialSkewMatrix )
```

Arguments expectred by `M_SetCamera` are:

| | |
|---|---|
| close | a pointer to a `CloseData` structure |
| NormalCameraMatrix | a pointer to a matrix that describes the camera's position in world space |
| SpecialSkewMatrix | a pointer to a specially formatted matrix that contains persprctive and screen projection information. You can create this matrix by calling `Matrix_Perspective`, described under the next heading. |

## The Matrix_Perspective Function

The `Matrix_Perspective` function creates a specially formatted skew matrix using a basic description of the frame buffer geometry and viewing frustum.

The syntax of the `Matrix_Perspective` function is,

```
void Matrix_Perspectiv( Matrix *specialSkewMatrix, ViewPyramid
    *view, float screenXMin, float screenXMax, float screenYMin,
    float screenYMax, float scaleW )
```

Arguments expected by `Matrix_Perpective` are:

| | |
|---|---|
| specialSkewMatrix | a pointer to a Matrix that is used to return the calculated matrix |
| View | a pointer to a view pyramid. (The view pyramid describes the basic view frustum; (see the *vp.* calls in Example 2-3 on Page 41.) |
| ScreenXMin | the minimum screen *X* position |
| ScreenXMax | the maximum screen *X* position |
| ScreenYMin | the minimum screen *Y* position |
| ScreenYMax | the maximum screen *Y* position |
| ScaleW | a scaler for *w*. |

## How a Skew Matrix Works

In Mercury, a skew matrix is implemented as an object called a `Matrix` structure. a Matrix structure is simply a code representation of a skew matrix, which can be represented as follows in a Mercury program:

```
Matrix camMtx = {
{
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    0.0, 0.0, 0.0
}
```

Each matrix created using the $F = CS$ equation is a 4-by-4 matrix. Internally, however, Mercury represents each such matrix as a 3-by-4 matrix. To reduce a 4-by-4 matrix to a 3- by-4 matrix, you simply drop a column. Usually, the dropped column is the fourth column, which corresponds to the homogeneous coordinate *w*. However, in perspective operations, the third column, representing *z*, is dropped instead, and the homogeneous coordinate is kept in order to perform the perspective.

When a point is transformed in world coordinates $(x, y, z, 1)$ by $F$ (the camera skew matrix in the preceding equation), the result is a screen coordinate $(x, y)$ ready for the M2 triangle engine and a $w$ value the reciprocal of which is sent to the triangle engine.

## A Camera Matrix in World Space

A camera matrix in world space is created like any other transformation in world space. For example, in the *helloworld* program, the following code positions and orients the camera:

```
Matrix_SetTranslationByVector(&camMtx, camLocation );
Vector3D_Zero( &lookAtPt );

Matrix_LookAt(&camMtx, &lookAtPt, camLocation, 0.0);
```

In this code fragment, the `Matrix_SetTranslationByVector` function positions the camera at `camLocation`, and the `Matrix_LookAt` function orients the camera to look at the origin (0, 0, 0). (For more details on the operation of these functions, see the appendices in this book and your Mercury online documentation).

## Constructing the Skew Matrix

The output of applying the camera transformation are coordinates $(x', y', z')$ which are oriented along the screen but need to be scaled to the resolution of the screen. In addition, the origin of the screen is the upper left hand corner and the screen's $y$ axis points downward. The skew matrix $S$ must be constructed by concatenating a perspective transform, $P$, with a scale and translate transformation T that produces screen coordinates.

$S = PT$

In this equation, the perspective transformation, $P$, provides the foreshortening of the scene in camera coordinates. It can be specified by the relationship between how far the camera is from the projection plane relative to the size of the projection plane. Figure 1-4 shows how such a perspective transformation can be built in terms of a view frustrum defined by the following quantities:

- $t$—the position of the center point relative to the top border of the projection plane
- $b$—the position of the center point relative to the bottom border of the projection plane
- $r$—the position of the center point relative to the right border of the projection plane
- $l$—the position of the center point relative to the left border of the projection plane
- $n$—the position of the projection plane from the eye point
- $x_{max}, y_{max}$ —the pixel coordinates of the bottom right hand corner of the screen,
- $x_{min}, y_{min}$ —the pixel coordinates of the top left hand corner of the screen.

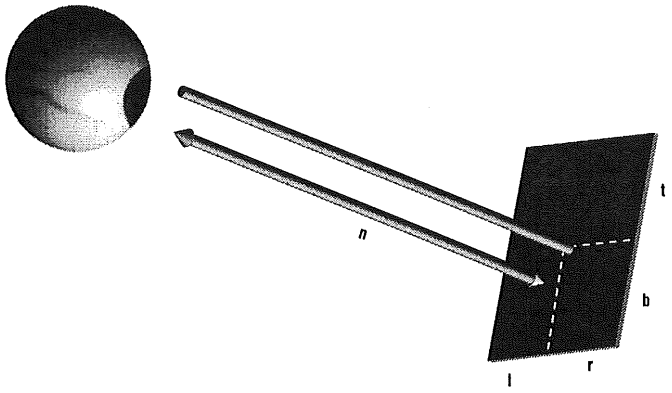


**Figure 1-4** *Building a perspective transformation.*

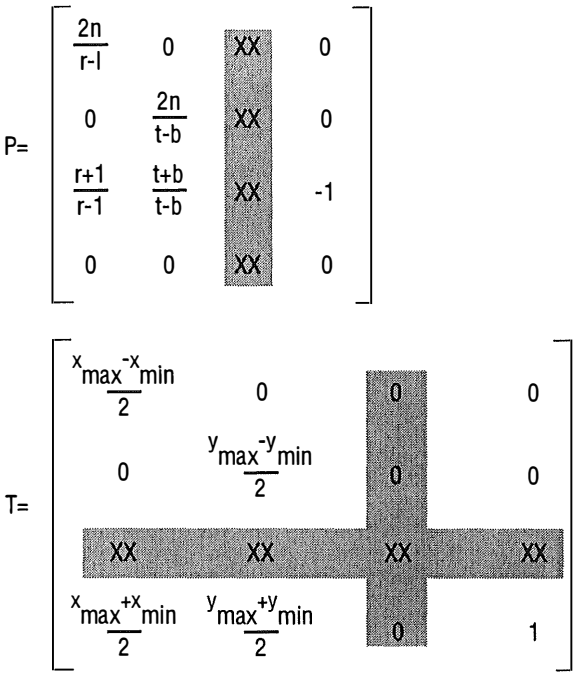


**Figure 1-5** *Illustrations of skew matrices.*

Figure 1-5 shows how the helper function `Matrix_Perspective` builds the skew matrix when the view pyramid data and the screen dimensions are passed to it. This function is passed a value of *1.0/hither* to scale the *w* value so that *w = 1.0* is at the hither plane.

In the illustratoin, the columns and rows that are deleted when the matrix is reduced from a 4-by-4 configuration to a 3-by-3 configuration (see "How a Skew Matrix Works" on page 31) appear against a gray background.

## Mercury Texture-Mapping

As explained in the *3DO Command List Toolkit* manual, M2 stores textures in a 16K block of RAM called *texture RAM*, or TRAM. Because of this texture-mapping architecture, the most efficient way to process different objects that access the same 16K of texture information is to process them at the same time. By processing textured objects in this way, you can avoid reloads of the same data into TRAM.

Mercury works very efficiently with the texture-mapping architecture of M2 because it provides a 16K block of TRAM that can contain many textures that are used at the same time. Moreover, Mercury sorts PODs in such a way that when multiple PODs need the same texture, they are processed at the same time, making multiple loads of TRAM unnecessary.

For instance, the *bigcircle* example provided on the M2 distribution media loads the TRAM 30 different times in a frame with two different kinds of textures:

- A 16K page containing three 8-bit textures (measuring 64 by 64 texels), mipmapped to four levels of detail
- A 16K page containing twelve 8-bit textures (measuring 32 by 32 texels), mipmapped to four levels of detail.

For more information on M2 texture mapping, see the *3DO Command List Toolkit* manual.

## Summary

This chapter introduced the components of the Mercury rendering engine, described their attributes, and provided some isolated examples showing how to use those components in an M2 application. In Chapter 2, "Using Mercury ," you'll see how Mercury can be used to create an actual application.

*Chapter 2*

# Using Mercury

At the beginning of Chapter 1, "Introducing Mercury, "you were presented with a list of basic steps for creating and building a game using the Mercury rendering engine. This chapter contains the same list of steps, described in considerably more detail.

To demonstrate the process of creating a Mercury application, this chapter uses the same sample application that was used to illustrate important points in Chapter 1: the *helloworld* example program that is provided on your M2 distribution CD-ROM.

This chapter is divided into the following sections:

## Anatomy of a Mercury Application

As noted in the section titled "Creating an Application with Mercury" on page 3, the following general steps are typically followed to create a game using Mercury:

1. Perform the initialization operations your application requires. Initialization of Mercury is a three-stage operation that requires the following steps:
   - Create and initialize a GState object
   - Initializing the Mercury rendering engine
   - Setting up and initializing a camera.

   Each of these operations is examined in more detail under a separate heading later in this chapter.
2. Perform the work that is needed to create and render each frame in your game. If you are using transparency, set the `srcaddr` field of your `CloseData` structure to the address of your current destination bitmap. Then write code that performs the standard per-frame operations, such as performing frame-buffering operations and page-flipping. This section of code also initializes the M2 destination blender.
3. Write the code that is required to run your game. In this section of code, you update all POD transforms and add new PODs to your game. Typically, at runtime, a game written using Mercury calls various Mercury and GState functions to perform such operations as:
   - Drawing models and backgrounds
   - Displaying rendered images
   - Updating camera positions as necessary
   - Checking bounding boxes of the objects used in your game
   - Determining what PODs should be displayed, and displaying them
   - Handling user input.
4. Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

The remainder of this chapter uses the helloworld program to describe and illustrate the preceding steps in much more detail.

## About the helloworld Program

The *helloworld* program is a bare-bones example of a Mercury application. It doesn't begin to demonstrate everything Mercury can do, but it is an excellent example of a basic framework that can be used as a starting point for creating a Mercury-based game.

Specifically, *helloworld* performs the following operations:

- Initializes a GState object, some bitmaps, and a display in preparation for rendering (in the *graphicsenv.c* file).
- Creates some simple geometry (in the *data.c* file) that is later used for two walls and a floor.
- Reads in a 3D model (in *filepod.c* and *bsdfreader.c*)
- Displays a single frame of data containing both a background POD and a model POD. The code that implements these operations appears in the program's main file, *helloworld.c*.

When you execute *helloworld*, you can position the camera anywhere you like using command-line arguments. Because the camera is aimed at the center of the input model's geometry, the scene that the program creates is always visible.

The program uses a standard texture reader (implemented in *tex_read.c*) and a standard SDF-file reader (implemented in *bsdf_read.c*). Both these readers can read data from the standard 3DO M2 graphics conversion tools.

When the user presses any control-pad button, the program terminates.

## Preparing to Write a Mercury Game

The first step in creating an application with Mercury is to perform the initialization operations that the application requires. Typically, Mercury initialization operations include the following steps:

1. Using the GState (Graphics State) calls provided in the M2 Graphics Portfolio, create and initialize a GState object. (For more information about GState objects, see the *3DO Command List Toolkit* manual.)
2. Using other function calls provided in the M2 Graphics Portfolio, set up a graphics environment for your application. (In the *helloworld* program, you can find the code that sets up a graphics environment in a file named *graphicsenv.c*. You'll get a chance to take a closer look at that file later in this chapter.)
3. Organize a set of data that defines the geometry that your application requires. Typically, this collection of data provides information that describes your application's matrix, vertex, lighting, and texturing specifications, along with any other data that Mercury may require to handle the PODs used by your application. (The *helloworld* program makes use of a data file named *data.c*.)
4. Load and initialize the data that is needed to create and render the 3D models you will be using in your application. Typically, this data includes texture, lighting, boundary, and scaling information, along with any other data that

Mercury requires to create and render the models used in your application. (Model information used by the *helloworld* example application is provided in a file named *filepod.c*.)

5. Initialize a `CloseData` structure including elements that won't change through out the title. (In Mercury, a structure called the `CloseData` structure contains all input to the graphics pipeline. For details, see "The CloseData Structure" on page 12.) To initialize a CloseData structure, you call the `M_Init` function, which initializes Mercury, and the `M_PackColor` function, which sets the values of the colors used in your application. (For more details on these function, see this book's appendices and your on-line Mercury documentation.) In the *helloworld* program, `M_Init` is called in a file named *helloworld.c*. You'll see how the `M_Init` function works in the *helloworld* example program.
6. Set up and initialize a camera for your application by following these steps: First, call the `Matrix_Perspective` function to initialize a skew matrix that includes perspective. Then initialize your camera by calling the `M_SetCamera` function (another function that is called in the *helloworld* program's *helloworld.c* file.)
7. Perform the lighting functions that your application requires. Call the `M_Prelight` function to prelight all pods that will be lighting using a pre-lit case:

```
M_Prelight(ppodsrc, ppoddst, pclosedata);
```

Each operation in the preceding list is described under a separate heading in the paragraphs that follow.

## Creating and Initializing a GState Object

When you start writing an application using Mercury, the first initialization operation you must perform is to create a GState object. Then you must set up and initialize a `CloseData` structure, and set up and initialize a camera.

In Mercury, the GState object—which is described in *3DO Command List Toolkit* manual—is a low-level object that all M2 libraries use to communicate with the M2 Triangle Engine. The GState object's main job is to encapsulate command list buffers using a format that all M2 libraries and folios can understand.

Mercury—like the Graphics Framework, the Graphics Pipeline, and all other M2 APIs—communicates with the Triangle Engine through the GState object. The GState objects provides Mercury with low-level routines that it can call to create command-list buffers and to send command-list buffers to the Triangle Engine for rendering into screen images.

Mercury uses the GState object to initialize the `CloseData` structure, a structure that contains all Mercury input to the graphics pipeline. (The `CloseData` structure is described in more detail in "The CloseData Structure" on page 12).

### Creating a GState Object with the GS_Create Function

A Mercury-based application, like any other M2 program, initializes a GState object by calling the Graphics Portfolio function `GS_Create`. In the *helloworld* application, `GS_Create` is called in the sequence of code shown in Example 2-1.

**Example 2-1** *Creating and Initializing a GState Object*

```
/* Create and set up the GState */
genv->gs = GS_Create();
if (genv->gs < 0) {
   retVal = Hg_NoMem;
   goto Failed_GSCreate;
}
err = GS_AllocLists(genv->gs, kNumCmdListBufs, kCmdListBufSize);
if (err < 0) {
   retVal = err;
   goto Failed_GSAllocLists;
}
GS_SetVidSignal(genv->gs, genv->d->signal);
GS_SetDestBuffer(genv->gs, genv->bitmaps[0]);

if (USE_Z_BUFFERING)
   GS_SetZBuffer(genv->gs, genv->bitmaps[genv->d->numScreens]);

/* Everything went OK */
retVal = 0;
printf("Using %d-bit display, size=%d x %d, %d bitmaps, %s\n",
genv->d->depth,genv->d->width,genv->d->height,genv->
   d->numScreens,
(USE_Z_BUFFERING ? "Z-buffered" : "No Z-buffer"));
goto Exit;
```

The code shown in Example 2-1 appears in a file named *graphicsenv.c*, which also performs a number of other functions that are used to set up a graphics environment for the *helloworld* program.

Two of those functions, `GS_SetVidSignal` and `GS_SetDestBuffer`, appear in Example 2-1. If you examine the *graphicsenv.c* file, you'll see other functions that perform important operations such as allocating memory for graphics, creating and displaying a view, and loading and displaying bitmaps.

The *3DO Command List Toolkit* manual provides more detailed information about GState objects and how they are used in M2 programs.

### Initializing the Mercury Rendering Engine

When your application has created and initialized a GState object, you're ready to initialize the Mercury rendering engine.

To initialize Mercury, you create and set up a structure called a `CloseData` structure. `CloseData` is a very important structure in Mercury because it contains all Mercury input to the graphics pipeline. `CloseData` is also used for temporary storage of intermediate calculations and temporary data.

To set up a `CloseData` structure, you call the Mercury function `M_Init`. (Appendix A, "Parts of the Mercury Engine," describes the syntax of the `M_Init` function.)

## Initializing Mercury in the helloworld Program

Once a `CloseData` structure is set up, the application can set important fields in that structure by simply using the C assignment operator (=). The *helloworld* program sets up a `CloseData` structure in Example 2-2, which you can find in the *helloworld.c* file:

**Example 2-2** *Initializing Mercury*

```
/* Initialize Mercury */
maxPodVerts = (maxPodVerts + 3) & ~3;
appData->close = M_Init(maxPodVerts, kMaxCmdListWords, genv->gs);
if (appData->close == NULL) {
   printf("Couldn't init Mercury.  Exiting\n");
   exit(1);
}
appData->close->fwclose = 1.01;
appData->close->fwfar = 100000.0;
appData->close->fscreenwidth = genv->d->width;
appData->close->fscreenheight = genv->d->height;
appData->close->depth = genv->d->depth;
```

In Example 2-2, the `CloseData` structure that is being set up is named `close`. It is referred to as `appData->close` because it is nested inside a larger structure named `appData`. In *helloworld*, `appData` a structure that contains both a `CloseData` structure and a `Matrix` structure. A `Matrix` structure—as you may recall from Chapter 1, "Introducing Mercury"—is a structure used to describe the location of a graphics object (such as a camera, pod, or the like).

The `CloseData` structure is described in more detail in "The CloseData Structure" on page 12. Under the next heading, "Creating and Setting Up a Camera," you'll learn more about how *helloworld* uses the `Matrix` structure.

### Creating and Setting Up a Camera

The last step in the Mercury initialization process is to create and set up a camera. Set up and initialize a camera for your application by following these steps:

1. Create and initialize a special kind of matrix called a *camera skew matrix*—that is, a matrix that includes perspective.
2. Call the `Matrix_Perspective` function to initialize camera skew matrix—that is, a skew matrix that includes perspective.
3. Initialize your camera by calling the `M_SetCamera` function.

In Mercury, the first step in setting up a camera is to create a camera skew matrix (for details, see "Camera Operations in Mercury" on page 30).

A camera skew matrix can be constructed using a Matrix data structure. In Mercury, a Matrix structure is simply a code representation of a transformation matrix. For example, the camera skew matrix used in the *helloworld* program is defined as follows in the *helloworld.c* file:

```
Matrix camMtx = {
{
   1.0, 0.0, 0.0,
   0.0, 1.0, 0.0,
   0.0, 0.0, 1.0,
   0.0, 0.0, 0.0
}
```

When you have created a camera skew matrix, you can set up a camera using a code sequence that looks something like the one shown in Example 2-3 (this code sequence is also taken from the *helloworld* program's *helloworld.c* source file.)

**Example 2-3** *Creating and Setting Up a Camera*

```
/* Set up the camera */

appData->skewMatrix = AllocMem(sizeof(Matrix), MEMTYPE_NORMAL);
if (appData->skewMatrix == NULL) {
   printf("Couldn't allocate skew matrix.  Exiting\n");
   exit(1);
     }

vp.left = -1.0;
```

```
vp.right = 1.0;
vp.top = 0.75;   /* Basic 4/3 aspect ratio, with positive Y UP */
vp.bottom = -0.75;
vp.hither = 1.2;

Matrix_Perspective(appData->skewMatrix, &vp, 0.0,
    (float)(genv->d->width), 0.0, (float)(genv->d->height),
    1.0/gHitherPlane);

Matrix_SetTranslationByVector(&camMtx, camLocation );
Vector3D_Zero( &lookAtPt );

Matrix_LookAt(&camMtx, &lookAtPt, camLocation, 0.0);

M_SetCamera( appData->close, &camMtx, appData->skewMatrix );
```

The code shown in Example 2-3 allocates memory for a camera skew matrix, sets fields in a `ViewPyramid` object named `vp`, and then performs some perspective-related operations by calling a series of functions named `Matrix_Perspective`, `Matrix_SetTranslationByVector`, `Vector3D_Zero`, and `Matrix_LookAt`. Finally, the application's camera is initialized with the call to `M_SetCamera` shown in the last line of the listing. (For more information about the `M_SetCamera` function, see "The M_SetCamera Function" on page 30.)

To learn more about the ViewPyramid object and the other function calls shown in Example 2-3, refer to this book's appendices and your on-line M2 documentation.

## Creating and Rendering Frames

When you have set up a camera for your application, you are ready to perform the work that is needed to create and render each frame in your title. At this stage of preprocessing, your application's game code calls various Mercury functions and Command List Toolkit (CLT) functions. to perform such operations as setting up the M2 destination blender, clearing the screen, and setting Z-buffers.

If you are using transparency, set the `srcaddr` field of your `CloseData` structure to the address of your current destination bitmap (`dstBM` in the following example):

```
pclosedata->srcaddr = dstBM->bm_Buffer;
```

Next, perform the usual per-frame operations, such as clearing the frame buffer and dispatching initialization commands. Typically, Mercury applications clear the frame buffer by calling the Command List Toolkit function

CLT_ClearFrameBuffer, and then initialize the M2 destination blender by calling the Mercury function M_DBInit:

```
CLT_ClearFrameBuffer(gs, .5, .5, .5, 1., 1, 1);
   M_DBInit(GS_Ptr(gs), 0,0,320, 240);
```

In the *helloworld* program, the CLT_ClearFrameBuffer and M_DBInit functions are called in a function named Program_Begin, which you can find in the *helloworld.c* file.

## Writing Your Game Code

When you've finished writing whatever frame-buffering code your title requires, you write your game code. In this section of your program, you update all POD transforms and add new PODs to your game.

Typically, your application's game code calls a few more Mercury and GState functions to perform such operations as:

- Drawing models and backgrounds
- Sending the command list to the Triangle Engine
- Checking bounding boxes of the objects used in your game
- Displaying PODs
- Handling user input.

Also, each time you update the position your application's camera, you must compute a camera matrix and combine it with a screen skew matrix:

```
pclosedata->fcamx = 0;
pclosedata->fcamy = -400;
pclosedata->fcamz = -600;
cameramat.Matrix_Data[i][j] = ...
Matrix_Mult((pMatrix)&pclosedata->fcamskewmatrix, &cameramat,
   &skewmat);
```

### Performing Bounds-Testing

Optionally, you can test the bounds of object groups to see which group of objects are visible:

```
M_BoundsTest(pbox, pclosedata);
```

After performing this test, you can check each bounding box and see if it has been found to be visible. If so, you must add any extra PODs that you need to the list of PODs in your ppodbuffer.

## Displaying Your Game's PODs

Finally, you keep track of the PODs used in your game and display them as needed. In the *helloworld* program, this is the section of code that displays the program's model and background pods (named `firstPod` and `gCornerPod`, respectively):

```
/* Draw the model */
firstPod = M_Sort(modelPodCount, gBSDF->pods, NULL);
M_Draw(firstPod, app->close);

/* Now draw the background. We could have just appended them
   together above, but this is how to do it when you need to
   render multiple pod lists. */

M_Draw(&gCornerPod, app->close);
```

When that's all done, you can swap your frame buffers.

## The Game-Play Section of the helloworld Program

In the *helloworld* program, all game code appears in a function named `Program_Begin`. (Yes, that's the same `Program_Begin` function that contains the `CLT_ClearFrameBuffer` and `M_DBInit` functions described in the preceding section, "Creating and Rendering Frames").

Example 2-4 on page 44 is a complete source-code listing of the complete `Program_Begin` function. Examine the comments in the listing carefully; they explain exactly how the action segment of the *helloworld* program works, all the way from setting up the M2 destination blender to responding to the user's control commands.

**Example 2-4** *The Program_Begin Function*

```
void Program_Begin(GraphicsEnv* genv, AppData* app,
   uint32 modelPodCount)
{
ControlPadEventData cped;
Pod                 *firstPod;

/* First, set up DBlender so that the clear will also clear
   Z, etc. */

GS_Reserve(genv->gs, M_DBInit_Size);
M_DBInit(GS_Ptr(genv->gs), 0, 0, genv->d->width,
```

```
    genv->d->height);

/* Clear screen and Z-buffer */
CLT_ClearFrameBuffer(genv->gs, 0.9, 0.9, 0.92, 1.0, TRUE, TRUE);

/* Now reset state, since CLT_ClearFB changed the state of
    the DBlender */

GS_Reserve(genv->gs, M_DBInit_Size);
M_DBInit(GS_Ptr(genv->gs), 0, 0, genv->d->width,
    genv->d->height);

/* Draw the model */
firstPod = M_Sort(modelPodCount, gBSDF->pods, NULL);
M_Draw(firstPod, app->close);

/* Now draw the background.  We could have just appended them
    together above, but this is how to do it when you need to
    render multiple pod lists. */

M_Draw(&gCornerPod, app->close);

/* Here's where the cmd list is sent to the Triangle Engine */
genv->gs->gs_SendList(genv->gs);

/* Once it's rendered, show the bitmap */
GS_WaitIO(genv->gs);
ModifyGraphicsItemVA(genv->d->view, VIEWTAG_BITMAP,
    GS_GetDestBuffer(genv->gs), TAG_END);

printf("Hit anything on the Control Pad to exit...\n");

GetControlPad(1, TRUE, &cped);
}
```

## Cleaning Up

Write code that shuts your game down and performs all needed cleanup functions when the user terminates the program.

This is easy; call

```
M_End(pclosedata);
```

and you're done.

## Summary

This chapter showed you exactly how to write a simple Mercury program. Appendix A, "Parts of the Mercury Engine," describes the syntax of many Mercury function calls and provides more details about how they are used in Mercury programs. For still more information about the function calls, macros, and data structures used in Mercury, see this book's appendices and your on-line Mercury documentation.

*Appendix A*

# Parts of the Mercury Engine

This appendix lists and describes the function calls provided in the Mercury rendering engine, arranged by functionality. For more detailed descriptions of the functions listed in this appendix, see your online Mercury documentation.

This appendix is divided into the following major sections:



## The Core Functions

This section lists the four core function calls provided in the Mercury API.: `M_Init`, `M_End`, `M_Sort`, and `M_Draw`. These functions were introduced in the section headed "Mercury's Four Core Functions" on page 4.

### M_Init

`M_Init` initializes the system for rendering. It must be called once before any other Mercury functions are called. It allocates, initializes and returns a `CloseData` structure that can subsequently be used for rendering.

The syntax of the `M_Init` function is:

```
CloseData * M_Init (uint32 nverts, uint32 clistsize, GState *gs)
```

Arguments expected by `M_Init` are:

| | |
|---|---|
| nverts | the maximum number of vertices to be used in a POD. |
| clistsize | the maximum number of words to be used in a command list. |
| gs | the GState object used to send command lists to the TE. |

## M_End

The `M_End` function frees all memory previously allocated by the `M_Init function`, including the memory allocated to the `CloseData` structure initialized by `M_Init`. This is the syntax of the `M_End` function:

```
void M_End(CloseData *pclosedata)
```

`M_End` takes one parameter: `pclosedata`, a value returned by a previous call to `M_Init`.

## M_Sort

The `M_Sort` function sorts the linked list of PODs passed in according to each field in the POD structure. The linked list of PODs that exists prior to a call to `M_Sort` may be arranged in a different order when `M_Sort` returns. `M_Sort` returns a pointer to the first pod in the sorted list

The syntax of the `M_Sort` function is:

```
Pod * M_Sort (uint32 count, Pod *podlist, Pod **plastpod)
```

Arguments expected by `M_Sort` are:

| | |
|---|---|
| count | the number of PODs passed in. |
| podlist | the first POD in a linked list of PODs. |
| plastpod | pointer to a pointer to the last POD <RETURNED>. |

## M_Draw

The `M_Draw` function takes a linked list of PODs as inputs, and outputs the corresponding M2 command list for rendering the PODs. It is the responsibility of your application to send all other M2 command lists for setting up the state of the M2 Triangle Engine state and for producing any effects. The command list produced by `M_Draw` consists of copies of command lists that setup the triangle engine for a case, load or select textures, supplied in the POD data structures, and vertex commands for drawing triangles.

`M_Draw` returns the number of triangles that actually are drawn to the screen. This number does not include rejected triangles. But it does include triangles created as a result of clipping.

The syntax of the `M_Draw` function is:

```
uint32 M_Draw (Pod *firstpod, CloseData *pclose)
```

Parameters expected by the `M_Draw` function are:

| | |
|---|---|
| firstpod | The first pod in a linked list of PODs in which all the changed flags are set as per M_Sort. |

| | |
|---|---|
| pclose | A CloseData area which has been properly initialized. |

## Utility Functions

Along with the four core functions described under the previous heading, Mercury provides three utility functions that perform lighting, bounds-testing, and texture-related operations.

This section describes each of Mercury's utility functions under a separate heading.

### M_Prelight

For each POD in a linked list provided to Mercury, the `M_PreLight` function computes the *rgb* values that result from the lighting calculation of the affected POD and places the result in a corresponding POD in your application's destination list. The source POD provided to the `M_Prelight` function must have normal vectors in its geometry data fields. When the `M_Prelight` function returns, the destination POD has *rgb* values in its geometry data fields.

The source and destination POD list can be the same. However, if you prelight the same geometry twice, the normals that are now colors will yield unexpected results.

This is the syntax of the `M_Prelight` function:

```
void M_PreLight (Pod* src, Pod* dst, CloseData* pclose)
```

Arguments expected by `M_Prelight` are:

| | |
|---|---|
| src | source POD list |
| dst | destination POD list |
| pclose | CloseData structure to use. |

### M_BoundsTest

The `M_BoundsTest` function tests each bounding box in a list of boxes and sets the flags field to indicate whether the box is visible or not. This is the syntax of the M_BoundsTest function:

```
void M_BoundsTest (BBoxList* bbox, CloseData* pclose)
```

`M_BoundsTest` expects the following parameters:

| | |
|---|---|
| bbox | A list of bounding boxes to test |
| pclose | CloseData to use |

### M_LoadPodTexture

By default, the texture reader places the `M_LoadPodTexture` in the `proutine` field of a `PodTexture` structure to load a texture. If you want to override this function, you can replace this function with your own.

```
typedef struct PodTexture
{
     void (*proutine)(void);
     struct TpageSnippets *ptpagesnippets;
     uint32 *ptexture;
     uint32 texturecount;
```

```
    uint32 texturebytes;
    uint32 *ppip;
    uint32 pipbytes;
} PodTexture;
```

### M_SetCamera

`M_SetCamera` takes a normal camera matrix in world space and multiplies it with a specially formatted skew matrix to form Mercury's internal camera skew matrix.

The syntax of the `M_SetCamera` function is:

```
void M_SetCamera( CloseData *close, Matrix *normalCameraMatrix,
   Matrix *specialSkewMatrix )
```

Arguments expected by `M_SetCamera` are:

| | |
|---|---|
| close | a pointer to Mercury's close data structure |
| NormalCameraMatrix | a pointer to a matrix that describes the camera's position in world space |
| SpecialSkewMatrix | a pointer to a specially formatted matrix that contains perspective and screen projection information. You can create this matrix by calling `Matrix_Perspective` (see next entry). |

### Matrix_Perspective

The `Matrix_Perspective` function creates a specially formatted skew matrix using a basic description of the frame buffer geometry and viewing frustum.

The syntax of the `Matrix_Perspective` function is,

```
void Matrix_Perspective( Matrix *specialSkewMatrix, ViewPyramid
   *view, float screenXMin, float screenXMax, float screenYMin,
   float screenYMax, float scaleW )
```

Arguments expected by `Matrix_Perpective` are:

| | |
|---|---|
| specialSkewMatrix | a pointer to a Matrix that is used to return the calculated matrix |
| View | a pointer to a view pyramid. (The view pyramid describes the basic view frustum; (see the *vp.* calls in Example 2-3 on Page 41.) |
| ScreenXMin | the minimum screen *X* position |
| ScreenXMax | the maximum screen *X* position |
| ScreenYMin | the minimum screen *Y* position |
| ScreenYMax | the maximum screen *Y* position |
| ScaleW | a scaler for *w*. |

## M_Draw. . . Functions

The name of each draw function used in Mercury begins with the letters `M_Draw`. For example, the `M_DrawDynLit` function dynamically lights triangles according to the list of lights in each POD. (In this manual, this set of functions is referred to generically as Mercury's `M_Draw. . .` functions. This group of function's should

be distinguished from Mercury's main `M_Draw` function, which is not followed by an ellipsis (. . .) when it appears in this volume.)

This section lists and describes Mercury's `Draw. . .` functions.

### M_DrawDynLit

The `M_DrawDynLit` function dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing *x, y, r, g, b,* and *w.*

The output alpha of the `M_DrawDynLit` function can represent transparency or fog.

### M_DrawDynLitTex

`M_DrawDynLitTex` dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing *x, y, r, g, b, a, u, v,* and *w.*

The output alpha can represent transparency or fog.

### M_DrawDynLitTrans

The `M_DrawDynLitTrans` function dynamically lights triangles according to the list of lights in each POD. It outputs triangle commands with vertices containing *x, y, r, g, b, a, u, v,* and *w.*

The output alpha represents transparency. The output u represents fog.

### M_DrawPreLit

The `M_DrawPreLit` function does not light triangles. It obtains the lighting color that is precomputed and already present in the input data. It outputs triangle commands with vertices containing *x, y, r, g, b,* and *w.* The output alpha can represent transparency or fog.

### M_DrawPreLitTex

`M_DrawPreLitTex` is another function that does not light triangles. It obtains the lighting color already pre-computed from the input data. It outputs triangle commands with vertices containing *x, y, r, g, b, a, u, v,* and *w.* The output alpha can represent transparency or fog.

### M_DrawPreLitTrans

`M_DrawPreLitTran` is still another function that does not light triangles. It obtains the lighting color already pre-computed from the input data. It outputs triangle commands with vertices containing *x, y, r, g, b, a, u, v,* and *w.* The output alpha represents transparency. The output *u* represents fog.

## M_Light. . . Functions

The light functions in Mercury are multi-purpose routines. Besides using Mercury's light functions for lighting, you can use them to perform specific per-vertex functions such as creating fog. For more detailed information about how to use Mercury's multi-purpose lighting functions, see the section headed "Mercury Lighting" on page 24.

All the light functions used in Mercury have names that begin with the letters M_Light. For example, the M _LightFog function computes an alpha value that can be used for a fogging effect. (In this manual, these functions are referred to as Mercury's M_Light. . . functions.)

## Initializing Mercury's Light Functions

As explained in "Mercury Lighting" on page 24, Mercury's light functions work a little differently from the other kinds of functions implemented in the Mercury API. First you construct a light list (see "Light Lists" on page 25), and then you supply Mercury with two other kinds of data structures for each POD you want to light: a lighting structure that describes the kind of lighting you want to create for the POD, and a Material structure that describes the kind of material that is being illuminated. When you have provided these two structures, Mercury has all it needs to call the appropriate lighting functions.

To provide Mercury with a light list, you use a LightList structure (described in "Light Lists" on page 25). Each entry in a LightList structure is made up of two elements: a pointer to a function and a pointer to a second data structure that describes the kind of lighting being used by the corresponding function. This information is provided on a per-vertex basis. Each function that is referenced in a LightList branches to a subroutine that initializes the specified lights on a per-object basis. But you do not have to be concerned with these internal details when you construct a light list. All you have to do is provide Mercury with the name of the lighting function you want to use and the name of the light structure that is associated with the function you have selected. Mercury then accesses the lighting function and its associated light structure and performs all necessary lighting operations automatically.

## Mercury's Light Functions Listed and Described

Under the headings that follow, Mercury's lighting functions are listed and described.

### M_LightFog

The M_LightFog function computes an alpha value that is used for a fogging effect. The fog is a linear function on *w*, where 1.0 shows the object fully and 0.0 is completely fogged. Anything that is at or beyond fog far will be completely fogged. Anything closer than fognear is unfogged. To specify the fogging range, Mercury uses a pair of values called dist1 and dist2, which the user can calculate using the formula presented later in this section under the heading

**Input**: The input to the M_LightFog function is a LightFog structure:

```
typedef struct LightFog
{
   float dist1, dist2;
   }LightFog;
```

**Fields**: The fields of the LightFog structure are:

| | |
|---|---|
| dist1, dist2 | fog distances |

**Per-vertex processing**: In per-vertex processing, if the fog is 1 at fognear and is 0 at fogfar, the alpha value is computed as follows:

```
dist1 = hither/(fogfar - fognear)
dist2 = fognear/(fogfar - fognear)

prelimvalue = dist1 * w - dist2,
clamp(prelim value)
alpha -= prelimvalue
```

### M_LightDir

The `M_LightDir` function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

**Input**: The input to the `M_LightDir` function is a `LightDir` structure:

```
typedef struct LightDir
{
   float nx, ny, nz;
   Color3 lightcolor;
} LightDir;
```

**Fields**: The fields of the `LightDir` structure are:

| | |
|---|---|
| nx, ny, nz | light direction |
| lightcolor | light color |

**Per-vertex processing**: In per-vertex processing, the `M_LightDir` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
color += clamp(-(Nl•Nv))*lightcolor
```

### M_LightPoint

`M_LightPoint` simulates a point light source whose light decreases with distance. The light comes from a point, and declines based on inverse square of the distance. The light intensity is a negated dot product between the light direction and the vertex normal.

The `maxdist` value is determined by multiplying intensity times color depth per component times the brightest components value. For 32-bit color each component is 8 bits long. Consequently, you multiply by 256. If you use 16-bit mode with a light of color 0.5, 0.2, 0.2 and an intensity of 25000, `maxdist` equals 25,000 * 32 * 0.5.

**Input**: The input to the `M_LightPoint` function is a `LightPoint` structure:

```
typedef struct LightPoint
{
   float x, y, z;
   float maxdist;
   float intensity;
   Color3 lightcolor;
} LightPoint;
```

**Fields**: The fields of the `LightPoint` structure are:

| | |
|---|---|
| x, y, z | light position |
| maxdist | maximum distance light is visible from |
| intensity | light intensity |
| lightcolor | light color |

**Per-vertex processing**: In per-vertex processing, the `M_LightPoint` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

$$N_l = (P_l - P_v) / \text{len}(P_l - P_v)$$
$$\text{color} \mathrel{+}= \text{clamp}(N_l \cdot N_v) * (\text{intensity}/\text{distance}^2) * \text{lightcolor}$$

### M_LightSoftSpot

When you create lighting using the `M_LightSoftSpot` function, the light you create originates from a point, and radiates in the direction of a normalized vector. The light intensity declines linearly, based on the dot product between the light direction and the light source to object point normalized vector. The light intensity is a negated dot product between the light direction and the vertex normal.

The `maxdist` value is determined by multiplying intensity by color depth per component by the brightest component's value. For 32-bit color, each component is 8 bits long; consequently, you multiply by 256. If you use 16-bit mode with a light of color 0.5, 0.2, 0.2 and an intensity of 25,000, `maxdist` equals 25000 * 32 * 0.5.

**Input**: The input to the `M_LightSoftSpot` function is a `M_LightSoftSpot` structure:

```
typedef struct LightSoftSpot
{
   float x, y, z;
   float nx, ny, nz;
   float maxdist;
   float intensity;
   float cos,invcos;
   Color3 lightcolor;
} LightSoftSpot;
```

**Fields**: The fields of the `M_LightSoftSpot` structure are:

| | |
|---|---|
| x, y ,z | light position |
| nx, ny, nz | light direction |
| maxdist | maximum distance light is visible from |
| intensity | light intensity |
| cos, invcos | determine the start and end of fadeout region |
| lightcolor | light color |

**Per-vertex processing**: In per-vertex processing, the `M_LightSoftSpot` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
angle1 = cos(outer angle)
angle2 = 1.0/(cos(inner angle)- cos(outer angle))

Nl = (Pl - Pv) / len(Pl - Pv)
color += clamp((Nl•Nv)-cos)*invcos
   *(intensity/distance2)*lightcolor
```

### M_LightFogTrans

The `M_LightFogTrans` function computes a u coordinate value that is used for a fogging effect with a transparent nontextured object. The fog is a linear function on *w*, where 1.0 shows the object fully and 0.0 is completely fogged. The lighting calculation computes a *ucoord* value between 0 and 1 that is later scaled to be between 0 and 16.

**Input**: The input to the `M_LightFogTrans` function is a `LightFog` structure:

```
typedef struct LightFog
   {
        float dist1, dist2;
   } LightFog;

dist1,dist2fog distances
```

**Fields**: The fields of the `LightFog` structure are:

| | |
|---|---|
| dist1, dist2 | fog distances |

**Per-vertex processing**: In per-vertex processing, the `M_LightFogTrans` function computes a u coordinate which is used for a fogging effect.

If the fog is 1 at `fognear` and 0 is at `fogfar`, the alpha value is computed as follows:

```
dist1 = hither/(fogfar - fognear)
dist2 = fognear/(fogfar - fognear)

prelim value = dist1 * w - dist2,
     which is 0 at fognear and 1 at fogfar
clamp prelim value to 0.0
ucoord -= prelim value
```

### M_LightDirSpec

The `M_LightDirSpec` function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

`M_LightDirSpec` also adds a specular reflection to each vertex. It does this by adding a calculated specular color to the pixel's color. The object's specular

properties are described in the object's material properties. The specularFLAG must be set in the POD data structure to inform the initialization code that it needs to transform the camera into the object's local coordinates.

**Input**: The input to the M_LightDirSpec function is a LightDire structure:

```
typedef struct LightDir
{
   float nx, ny, nz;
   Color3 lightcolor;
} LightDir;
```

**Fields**: The fields of the LightDir structure are:

| | |
|---|---|
| nx, ny, nz | light direction |
| lightcolor | light color |

**Per-vertex processing**: In per-vertex processing, the M_LightDirSpec function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
color += clamp(-(Nl•Nv))*lightcolor
Nc = (Pv - Pc)/ len(Pv - Pc)
Nr = ((2*(Nl•Nv))*Nv)-Nl
Ns = Nc•Nr
specular = Ns/(shine-shine*Ns+Ns)
color += specular * specularcolor
```

### M_LightDirSpecTex

The M_LightDirSpecTex function approximates illumination by a light source located at infinity (such as the sun) for which all points are equidistant and only the direction predominates in the calculation. The light intensity is a negated dot product between the light direction and the vertex normal.

M_LightDirSpecTex also adds a specular reflection to each vertex. It does this by adding a constant specular color, scaled by alpha, to the pixel's color in the destination blender. The object's specular properties are described in the object's material properties. The specularFLAG must be set in the POD data structure to inform the initialization code that it needs to transform the camera into the object's local coordinates.

**Input**: The input to the M_LightDirSpecTex function is a LightDir structure:

```
typedef struct LightDir
{
   float nx, ny, nz;
   Color3 lightcolor;
} LightDir;
```

**Fields**: The fields of the LightDir structure are:

| | |
|---|---|
| nx, ny, nz | light direction |
| lightcolor | light color |

**Per-vertex processing**: In per-vertex processing, the `M_LightDirSpecTex` function calculates the vertex color and adds it to the accumulators. The value is calculated as follows:

```
color += clamp(-(Nl•Nv))*lightcolor
Nc = (Pv - Pc)/ len(Pv - Pc)
Nr = ((2*(Nl•Nv))*Nv)-Nl
Ns = Nc•Nr
specular = Ns/(shine-shine*Ns+Ns)
alpha += specular
```

## Texture-Blender and Destination-Blender Setup Macros

In Mercury, the command lists used by a case are assembled using a set of macros that are designed to set up the Texture Blender and the Destination Blender. These macros are collectively referred to as command-list setup macros.

When you call Mercury's texture-blender setup macros and destination-blender setup macros, the inputs to the Texture Blender are the Gouraud-interpolated colors and alpha calculated using a primitive, a supplied texture color, a supplied alpha, and a constant. The inputs to the Destination Blender are the outputs from the Texture Blender, a source frame buffer and a constant.

This section lists and describes each of Mercury's command-list setup macros.

### M_TBNoTex

When you call the `M_TBNoTex` macro, the Texture Blender outputs primitive color and alpha.

### M_TBLitTex

When your application calls `M_TBLitTex`, the Texture Blender multiplies the primitive color by the looked up texture color. The primitive alpha is passed through unchanged.

### M_TBTex

When `M_TBTex` is called, the Texture Blender passes only looked up texture color and alpha.

### M_TBFog

When Mercury executes the `M_TBFog` macro, the Texture Blender linearly interpolates between primitive color and a constant holding the color of the fog. The u texture coordinate determines the amount of the blend.

### M_DBNoBlend

The `M_DBNoBlend` macro causes Mercury to disable all blending the Destination Blender. The resulting output is the output of the Texture Blender.

### M_DBFog

When `M_DBFog` is called, then Destination Blender linearly interpolates between the output from the Texture Blender and a constant holding the color of the fog. The alpha passed in from the Texture Blender determines the amount of the blend.

### M_DBSpec

When the `M_DBSpec` macro is executed, the Destination Blender adds the output from the Texture Blender to the product of the specular highlight color times the alpha value output from the Texture blender.

### M_DBTrans

When `M_DBTrans` is called, Destination Blender linearly interpolates between the output of the Texture Blender and a corresponding pixel in a source frame buffer. The alpha value passed in from the Texture Blender determines the amount of the blend.

### M_DBInit

The `M_DBInit` macro initializes the destination blender state for use with other M_DB calls. This must be called once per frame to set the hardware in a known state.

## Case Setup Functions

In Mercury, the *case* of a POD is the combination of the state in the triangle engine and the draw function used to construct the triangle commands. Each case used in Mercury is equipped with a small setup function that is called when the first POD of the case is encountered—that is, when the *samecaseFLAG* is not set in the POD. When a setup function is called, it simply places in the `CloseData` structure the draw function that all PODs used by, and then loads a command list into the Triangle Engine.

Mercury provides case functions for 15 predefined cases that can be obtained from combinations of the following prefixes:

| | |
|---|---|
| dynlit \| prelit | Two types of lighting (dynamic lighting or prelighting) |
| fog \| spec \| none | Three types of constant color blending (fog, specular, or none) |
| trans \| none | either transparent or opaque |
| tex \| none | either textured or non-textured. |

Some possible combinations—for example fog + trans + tex—are not valid because there is no corresponding configuration of the blending stages in the M2 hardware.

The setup functions used in Mercury are listed in Table A-1.

**Table A-1** *Combinations of Predefined Cases*

| Setup Function | Draw Function | Texture Blender Command | Destination Blender Command |
|---|---|---|---|
| M_SetupDynLit | M_DrawDynLit | M_TBNoTex | M_DBNoBlend |
| M_SetupDynLitTex | M_DrawDynLitTex | M_TBLitTex | M_DBNoBlend |
| M_SetupDynLitTrans | M_DrawDynLit | M_TBNoTex | M_DBTrans |
| M_SetupDynLitTransTex | M_DrawDynLitTex | M_TBLitTex | M_DBTrans |

| | | | |
|---|---|---|---|
| M_SetupDynLitFog | M_DrawDynLit | M_TBNoTex | M_DBFog |
| M_SetupDynLitFogTex | M_DrawDynLitTex | M_TBLitTex | M_DBFog |
| M_SetupDynLitFogTrans | M_DrawDynLitTrans | M_TBFog | M_DBTrans |
| M_SetupDynLitSpecTex | M_DrawDynLitTex | M_TBLitTex | M_DBSpec |
| M_SetupPreLit | M_DrawPreLit | M_TBNoTex | M_DBNoBlend |
| M_SetupPreLitTex | M_DrawPreLitTex | M_TBLitTex | M_DBNoBlend |
| M_SetupPreLitTrans | M_DrawPreLit | M_TBNoTex | M_DBTrans |
| M_SetupPreLitTransTex | M_DrawPreLitTex | M_TBLitTex | M_DBTrans |
| M_SetupPreLitFog | M_DrawPreLit | M_TBNoTex | M_DBFog |
| M_SetupPreLitFogTex | M_DrawPreLitTex | M_TBLitTex | M_DBFog |
| M_SetupPreLitFogTrans | M_DrawPreLitTrans | M_TBFog | M_DBTrans |

*Appendix B*

# Matrix Calls

This appendix provides a complete list of reference pages for each Matrix call. The calls included in this appendix, and a brief description, are listed below.

| | |
|---|---|
| Matrix_ | Computes a perspective matrix that is concatenated on to the camera matrix. |
| Matrix_BillboardX | Calculates a matrix that lays perpendicular to a vector about it's X-axis |
| Matrix_BillboardY | Calculates a matrix that lays perpendicular to a vector about it's Y-axis |
| Matrix_BillboardZ | Calculates a matrix that lays perpendicular to a vector about it's Z-axis |
| Matrix_Construct | Allocates an instance of a matrix and sets it to identity. |
| Matrix_Copy | Copyies from a source matrix to a detination matrix. |
| Matrix_CopyOrientation | Copy the orientation of a source matrix to a destination matrix. |
| Matrix_CopyTranslation | Copy the translation of a source matrix to a destination matrix. |
| Matrix_Destruct | Frees the givem matrix from memory. |
| Matrix_FullInvert | Full matrix invert. |
| Matrix_GetBank | Calculates the heading component of a matrix. |
| Matrix_GetElevation | Calculates the elevation component of a matrix. |
| Matrix_GetHeading | Calculates the heading component of a matrix. |
| Matrix_GetTranslation | Copy the transformation information from the matrix to a vector. |
| Matrix_GetpTranslation | Get a pointer to the translation within a matrix. |

| | |
|---|---|
| Matrix_GetTranslation | Sets the matrix to identity. |
| Matrix_Invert | Matrix invert. |
| Matrix_LookAt | Calculates a matrix that looks down the -z-axis at a specificed point. |
| Matrix_LookAt | Translate matrix by three floats in local orientation. |
| Matrix_MoveByVector | Translate matrix by vector in local orientation. |
| Matrix_Mult | Matrix multiply two source matrices. |
| Matrix_Multiply | Multiplies a source matrix with the destination matrix. |
| Matrix_MultiplyOrientation | Multiplies the orientation section of a source matrix with the destination matrix. |
| Matrix_MultOrientation | Matrix multiply the orientation section of two source matrices. |
| Matrix_Normalize | Normalize the given matrix. |
| Matrix_Print | Print the matrix. |
| Matrix_Rotate | Rotate the matrix in world space. |
| Matrix_RotateLocal | Rotate the orientation of a matrix in world space. |
| Matrix_RotateX | Rotate the orientation of a matrix in world space about the X-axis. |
| Matrix_RotateXLocal | Rotate the matrix in world orientation and in local space about the X-axis. |
| Matrix_RotateY | Rotate the matrix in world space about the X-axis. |
| Matrix_RotateYLocal | Rotate the matrix in world orientation and in local space about the Y-axis. |
| Matrix_RotateZ | Rotate the orientation of a matrix in world space about the Z-axis. |
| Matrix_RotateZLocal | Rotate the matrix in world orientation and in local space about the Z-axis. |
| Matrix_Scale | Scale matrix by three floats. |
| Matrix_ScaleByVector | Scales the matrix by a vector. |
| Matrix_ScaleLocal | Scale matrix in local coordinates by three floats. |
| Matrix_ScaleLocalByVector | Scale matrix in local coordinates by a vector. |
| Matrix_SetTranslation | Set the matrix's translation. |
| Matrix_SetTranslationByVector | Set a matrix's translation from a vector. |
| Matrix_Translate | Translate the matrix by three floats. |
| Matrix_TranslateByVector | Translate the matrix by a vector. |
| Matrix_Turn | Rotate the matrix in local space. |

| | |
|---|---|
| Matrix_TurnLocal | Rotate the matrix in local space updating the orientation only. |
| Matrix_TurnX | Rotate the matrix in local space about the X-axis. |
| Matrix_TurnXLocal | Rotate the matrix in local space about the X-axis updating the orientation only. |
| Matrix_TurnY | Rotate the matrix in local space about the Y-axis. |
| Matrix_TurnYLocal | Rotate the matrix in local space about the Y-axis updating the orientation only. |
| Matrix_TurnZ | Rotate the matrix in local space about the Z-axis. |
| Matrix_TurnZLocal | Rotate the matrix in local space about the Z-axis updating the orientation only. |
| Matrix_Zero | Zeros the contents of a matrix. |

# Matrix_

Computes a perspective matrix that is concatenated on to the camera matrix.

## Synopsis

```
void Matrix_Perspective(pMatrix matrix, pViewPyramid p, float
screen_xmin, float screen_xmax, float screen_ymin, float
screen_ymax, float wscale)
```

## Description

Computes a perspective matrix that is concatenated on to the camera matrix.

## Arguments

| | |
|---|---|
| matrix | Pointer to the destination Matrix structure. |
| p | Pointer to the:<br>screen_xmin value.<br>screen_xmax value.<br>screen_ymin value.<br>screen_ymax value.<br>wscale value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_BillboardX

Calculates a matrix that lays perpendicular to a vector about its X-axis

### Synopsis

```
void Matrix_BillboardX( Matrix *m, Vector3D *v );
```

### Description

Calculates a matrix that lays perpendicular to a vector about its X-axis

### Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| v | Pointer to a vector to 'lookat'. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_BillboardY

Calculates a matrix that lays perpendicular to a vector about its Y-axis

## Synopsis

```
void Matrix_BillboardY( Matrix *m, Vector3D *v );
```

## Description

Calculates a matrix that lays perpendicular to a vector about its Y-axis

## Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| v | Pointer to a vector to 'lookat'. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_BillboardZ

Calculates a matrix that lays perpendicular to a vector about its Z-axis

### Synopsis

```
void Matrix_BillboardZ( Matrix *m, Vector3D *v );
```

### Description

```
Calculates a matrix that lays perpendicular to a vector about
its Z-axis
```

### Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| v | Pointer to a vector to 'lookat'. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Construct

Allocates an instance of a matrix and sets it to identity.

## Synopsis

```
Matrix* Matrix_Construct( void );
```

## Description

Allocates an instance of a matrix and sets it to identity.

## Arguments

void

## Return Value

Matrix* Pointer to the new Matrix structure.

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_Copy

Copies from a source matrix to a detination matrix.

### Synopsis

```
void Matrix_Copy( Matrix *m, Matrix *a ); Matrix*
Matrix_Construct( void );
```

### Description

Copies from a source matrix to a detination matrix.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| a* | Pointer to the source Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_CopyOrientation

Copy the orientation of a source matrix to a destination matrix.

## Synopsis

```
void Matrix_CopyOrientation( Matrix *m, Matrix *b );
```

## Description

Copy the orientation of a source matrix to a destination matrix.

## Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| b | Pointer to a source matrix. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_CopyTranslation

Copy the translation of a source matrix to a destination matrix.

### Synopsis

```
void Matrix_CopyTranslation( Matrix *m, Matrix *b );
```

### Description

Copy the translation of a source matrix to a destination matrix.

### Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| b | Pointer to a source matrix. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_Destruct

Frees the givem matrix from memory.

### Synopsis

```
void Matrix_Destruct( Matrix *m );
```

### Description

Frees the givem matrix from memory.

### Arguments

| | |
|---|---|
| m* | Pointer to the Matrix structure to me deleted. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_FullInvert

Full matrix invert.

### Synopsis

```
Err Matrix_FullInvert(pMatrix dst, pMatrix src);
```

### Description

The code is based on the code presented in "Graphics Gems" (ed. Andrew Glassner), in "Matrix Inversion" (pg. 766). The code has been modified to work with the 3x4 matrices we use. We don't have a fourth column, so it is hardwired as (0, 0, 0, 1). Print the matrix.

### Arguments

| | |
|---|---|
| dst | Pointer to the destination Matrix structure. |
| src | Pointer to the source Matrix structure. |

### Return Value

### Err

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_GetBank

Calculates the heading component of a matrix.

## Synopsis

```
float Matrix_GetBank( Matrix *m );
```

## Description

Calculates the banking component of a matrix.

## Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |

## Return Value

float angle in radians. 0 is aligned along the +x axis. Positive angles are to the righthand side up. Negative angles are to the righthand side down.

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_GetElevation

Calculates the elevation component of a matrix.

### Synopsis

```
float Matrix_GetElevation( Matrix *m );
```

### Description

Calculates the elevation component of a matrix.

### Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |

### Return Value

float angle in radians. 0 is aligned along the -z axis. Positive angles are to the up from the z axis. Negative angles are to the down from the z axis.

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_GetHeading

Calculates the heading component of a matrix.

## Synopsis

```
float Matrix_GetHeading( Matrix *m );
```

## Description

Calculates the heading component of a matrix.

## Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |

## Return Value

Float angle in radians. 0 is aligned along the -z axis. Positive angles are to the righthand side. Negative angles are to the lefthand side.

## Implementation

```
Version 1.0
```

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_GetTranslation

Copy the transformation information from the matrix to a vector.

### Synopsis

```
void Matrix_GetTranslation( Matrix *m, Vector3D *v );
```

### Description

Copy the transformation information from the matrix to a vector.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the destination Vector3D structure. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Construct(), Matrix_GetpTranslation(),
Matrix_SetTranslationByVector(), Matrix_SetTranslation(),
Matrix_TranslateByVector(), Matrix_Translate(),Matrix_Move()
```

# Matrix_GetpTranslation

Get a pointer to the translation within a matrix.

## Synopsis

```
Vector3D* Matrix_GetpTranslation( Matrix *m );
```

## Description

Get a pointer to the translation within a matrix.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |

## Return Value

| | |
|---|---|
| v* | Pointer to the matrix's translation value. |

## Implementation

```
Version 1.0
```

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Construct(), Matrix_GetTranslation(),
Matrix_SetTranslationByVector(), Matrix_SetTranslation(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move()
```

## Matrix_Identity

Sets the matrix to identity.

### Synopsis

```
Matrix_Identity( nMatrix *m );
```

### Description

Sets the matrix to identity.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Invert

Matrix invert.

## Synopsis

```
void Matrix_Invert( Matrix *dst, Matrix *src );
```

## Description

Matrix invert.

## Arguments

| | |
|---|---|
| dst | Pointer to the destination Matrix structure. |
| src | Pointer to the source Matrix structure. |

## Return Value

Err

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_LookAt

Calculates a matrix that looks down the -z-axis at a specificed point.

### Synopsis

```
void Matrix_LookAt(Matrix *m, Vector3D *pWhere, Vector3D
*pMe, float twist);
```

### Description

Calculates a matrix that lays perpendicular to a vector about its Y-axis

### Arguments

| | |
|---|---|
| m | Pointer to the destination Matrix structure. |
| pWhere | Pointer to a value to look at. |
| pMe | Pointer to a value to look from. |
| twist | A rotation in radians away from the up vector. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

# Matrix_Move

Translate matrix by three floats in local orientation.

## Synopsis

```
void nMatrix_Move( nMatrix *m, float x, float y, float z );
```

## Description

Translate matrix by three floats in local orientation.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| x | Move X value. |
| y | Move Y value. |
| z | Move Z value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_MoveByVector

Translate matrix by vector in local orientation.

### Synopsis

```
void nMatrix_MoveByVector( nMatrix *m, nVector3D *v );
```

### Description

Translate matrix by vector in local orientation.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Mult

Matrix multiply two source matrices.

## Synopsis

```
void nMatrix_Mult( nMatrix *m, nMatrix *a, nMatrix *b );
```

## Description

Matrix multiply two source matrices. [m =a x b ]

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| a* | Pointer to the first source Matrix structure. |
| a* | Pointer to the second source Matrix structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

<matrix.h>, mercury/lib

## See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(), Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(), Matrix_MultiplyOrientation(), Matrix_Zero(), Matrix_Identity(), Matrix_TranslateByVector(), Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(), Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(), Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(), Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

## Matrix_Multiply

Multiplies a source matrix with the destination matrix.

### Synopsis

```
void Matrix_Multiply( Matrix *m, Matrix *a );
```

### Description

Multiplies a source matrix with the destination matrix. [m =m x a ]

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| a* | Pointer to the source Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_MultiplyOrientation

Multiplies the orientation section of a source matrix with the destination matrix.

## Synopsis

```
void nMatrix_MultiplyOrientation( nMatrix *m, nMatrix *a );
```

## Description

Multiplies the orientation section of a source matrix with the destination matrix. [m =m x a ]

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| a* | Pointer to the source Matrix structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_MultOrientation

Matrix multiply the orientation section of two source matrices.

### Synopsis

```
void nMatrix_MultOrientation( nMatrix *m, nMatrix *a,
      nMatrix *b );
```

### Description

Matrix multiply the orientation section of two source matrices. [m =a x b ]

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| a* | Pointer to the first source Matrix structure. |
| b* | Pointer to the second destination Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Normalize

Normalize the given matrix.

## Synopsis

```
void Matrix_Normalize( Matrix *m );
```

## Description

Normalize the given matrix.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_Print

Print the matrix.

### Synopsis

```
void nMatrix_Print( nMatrix *m );
```

### Description

Print the matrix.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Rotate

Rotate the matrix in world space.

## Synopsis

```
void nMatrix_Rotate( nMatrix *m, char axis, float r );
```

## Description

Rotate the matrix in world space.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| axis | World axis to rotate around [ 'X','Y' or 'Z' ]. |
| r | Rotation angle in radians |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_RotateLocal

Rotate the orientation of a matrix in world space.

### Synopsis

```
void nMatrix_RotateLocal( nMatrix *m, char axis, float r );
```

### Description

Rotate the orientation of a matrix in world space.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| axis | World axis to rotate around [ 'X','Y' or 'Z' ]. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_RotateX

Rotate the orientation of a matrix in world space about the X-axis.

## Synopsis

```
void nMatrix_RotateX( nMatrix *m, float r );
```

## Description

Rotate the orientation of a matrix in world space about the X-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_RotateXLocal

Rotate the matrix in world orientation and in local space about the X-axis.

### Synopsis

```
void Matrix_RotateXLocal( Matrix *m, float r );
```

### Description

Rotate the matrix in world orientation and in local space about the X-axis

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateYLocal(), Matrix_RotateZLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()
```

# Matrix_RotateY

Rotate the orientation of a matrix in world space about the Y-axis.

## Synopsis

```
void nMatrix_RotateY( nMatrix *m, float r ); Matrix*
Matrix_Construct( void );
```

## Description

Rotate the orientation of a matrix in world space about the Y-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_RotateYLocal

Rotate the matrix in world orientation and in local space about the Y-axis.

### Synopsis

```
void Matrix_RotateYLocal( Matrix *m, float r );
```

### Description

Rotate the matrix in world orientation and in local space about the Y-axis

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateXLocal(), Matrix_RotateZLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()
```

# Matrix_RotateZ

Rotate the orientation of a matrix in world space about the Z-axis.

## Synopsis

```
void nMatrix_RotateZ( nMatrix *m, float r );
```

## Description

Rotate the orientation of a matrix in world space about the Z-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. r Rotation angle in radians |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_RotateZLocal

Rotate the matrix in world orientation and in local space about the Z-axis.

### Synopsis

```
void Matrix_RotateZLocal( Matrix *m, float r );
```

### Description

Rotate the matrix in world orientation and in local space about the Z-axis

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(),
Matrix_RotateZ(), Matrix_RotateLocal(),
Matrix_RotateXLocal(), Matrix_RotateYLocal(), Matrix_Turn(),
Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(),
Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal()
```

# Matrix_Scale

Scale matrix by three floats.

## Synopsis

```
void nMatrix_Scale( nMatrix *m, float x, float y, float z );
Matrix* Matrix_Construct( void );
```

## Description

Scale matrix by three floats.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| x | Scale X value. |
| y | Scale y value. |
| z | Scale Z value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

<matrix.h>, mercury/lib

## See Also

Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(), Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(), Matrix_MultiplyOrientation(), Matrix_Zero(), Matrix_Identity(), Matrix_TranslateByVector(), Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(), Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(), Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(), Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(), Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(), Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()

## Matrix_ScaleByVector

Scales the matrix by a vector.

### Synopsis

```
void nMatrix_ScaleByVector( nMatrix *m, nVector3D *v );
```

### Description

Scales the matrix by a vector.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_ScaleLocal

Scale matrix in local coordinates by three floats.

## Synopsis

```
void Matrix_ScaleLocal( Matrix *m, float x, float y, float z
);
```

## Description

Scale matrix in local coordinates by three floats.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| x | Scale x value. |
| y | Scale y value. |
| z | Scale z value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

<matrix.h>, mercury/lib

## See Also

Matrix_ScaleByVector(), Matrix_Scale(), Matrix_ScaleLocalByVector()

## Matrix_ScaleLocalByVector

Scale matrix in local coordinates by a vector.

### Synopsis

```
void Matrix_ScaleLocalByVector( Matrix *m, Vector3D *v );
```

### Description

Scale matrix in local coordinates by a vector.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

<matrix.h>, mercury/lib

### See Also

Matrix_ScaleByVector(), Matrix_Scale(), Matrix_ScaleLocal()

# Matrix_SetTranslation

Set the matrix's translation.

## Synopsis

```
void Matrix_SetTranslation(Matrix *m, float x, float y, float
z);
```

## Description

Set the matrix's translation.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| x | New x value. |
| y | New y value. |
| z | New z value. |

## Return Value

void

## Implementation

```
Version 1.0
```

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Construct(), Matrix_GetTranslation(),
Matrix_GetpTranslation(), Matrix_SetTranslationByVector(),
Matrix_TranslateByVector(), Matrix_Translate(),
Matrix_MoveByVector(), Matrix_Move()
```

## Matrix_SetTranslationByVector Set a matrix's translation from a vector.

### Synopsis

```
void Matrix_SetTranslationByVector( Matrix *m, Vector3D *v );
```

### Description

Set a matrix's translation from a vector.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

```
Version 1.0
```

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Construct(), Matrix_GetTranslation(),
Matrix_GetpTranslation(), Matrix_SetTranslation(),
Matrix_TranslateByVector(), Matrix_Translate(),
Matrix_MoveByVector(), Matrix_Move()
```

# Matrix_Translate

Translate the matrix by three floats.

## Synopsis

```
void nMatrix_Translate( nMatrix *m, float x, float y,
      float z );
```

## Description

Translate the matrix by three floats.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| x | Translate X value. |
| y | Translate Y value. |
| Z | Translate Z value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_TranslateByVector

Translate the matrix by a vector.

### Synopsis

```
void nMatrix_TranslateByVector( nMatrix *m, nVector3D *v );
```

### Description

Translate the matrix by a vector.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| v* | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Turn

Rotate the matrix in local space.

## Synopsis

```
Matrix_Turn( nMatrix *m, char axis, float r );
```

## Description

Rotate the matrix in local space.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| axis | World axis to rotate around [ 'X','Y' or 'Z' ]. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_TurnLocal

Rotate the matrix in local space updating the orientation only.

### Synopsis

```
void nMatrix_TurnLocal( nMatrix *m, char axis, float r );
```

### Description

Rotate the matrix in local space updating the orientation only.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| axis | World axis to rotate around [ 'X','Y' or 'Z' ]. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_TurnX

Rotate the matrix in local space about the X-axis.

## Synopsis

```
void nMatrix_TurnX( nMatrix *m, float r );
```

## Description

Rotate the matrix in local space about the X-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_TurnXLocal

Rotate the matrix in local space about the X-axis updating the orientation only.

### Synopsis

```
void nMatrix_TurnXLocal( nMatrix *m, float r );
```

### Description

Rotate the matrix in local space about the X-axis updating the orientation only.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_TurnY

Rotate the matrix in local space about the Y-axis.

## Synopsis

```
void nMatrix_TurnY( nMatrix *m, float r ); Matrix*
Matrix_Construct( void );
```

## Description

Rotate the matrix in local space about the Y-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_TurnYLocal

Rotate the matrix in local space about the Y-axis updating the orientation only.

### Synopsis

```
void nMatrix_TurnYLocal( nMatrix *m, float r );
```

### Description

Rotate the matrix in local space about the X-axis updating the orientation only.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_TurnZ

Rotate the matrix in local space about the Z-axis.

## Synopsis

```
void nMatrix_TurnZ( nMatrix *m, float r ); Matrix*
Matrix_Construct( void );
```

## Description

Rotate the matrix in local space about the Z-axis.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

## Matrix_TurnZLocal

Rotate the matrix in local space about the Z-axis updating the orientation only.

### Synopsis

```
void nMatrix_TurnZLocal( nMatrix *m, float r );
```

### Description

Rotate the matrix in local space about the Z-axis updating the orientation only.

### Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |
| r | Rotation angle in radians. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

# Matrix_Zero

Zeros the contents of a matrix.

## Synopsis

```
void nMatrix_Zero( nMatrix *m );
```

## Description

Zeros the contents of a matrix.

## Arguments

| | |
|---|---|
| m* | Pointer to the destination Matrix structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Matrix_Destruct(), Matrix_Normalize(), Matrix_Copy(),
Matrix_Mult(), Matrix_Multiply(), Matrix_MultOrientation(),
Matrix_MultiplyOrientation(), Matrix_Zero(),
Matrix_Identity(), Matrix_TranslateByVector(),
Matrix_Translate(), Matrix_MoveByVector(), Matrix_Move(),
Matrix_ScaleByVector(), Matrix_Scale(), Matrix_Rotate(),
Matrix_RotateX(), Matrix_RotateY(), Matrix_RotateZ(),
Matrix_Turn(), Matrix_TurnX(), Matrix_TurnY(),
Matrix_TurnZ(), Matrix_TurnLocal(), Matrix_TurnXLocal(),
Matrix_TurnYLocal(), Matrix_TurnZLocal(), Matrix_Print()
```

*Appendix C*

# Vector Calls

This appendix provides a complete list of reference pages for each Matrix call. The calls included in this appendix, and a brief description, are listed below.

| | |
|---|---|
| Vector3D_Add | Add a source vector to a destination vector. |
| Vector3D_Average | Calculated the average of two vectors. |
| Vector3D_Compare | Compare two source vectors. |
| Vector3D_CompareFuzzy | Compare two source vectors with fuzzy edges. |
| Vector3D_Construct | Creates an instance of Vector3D. |
| Vector3D_Copy | Copy a source vector to a destination vector. |
| Vector3D_Cross | Calculates the cross product between two vectors. |
| Vector3D_Destruct | Deletes a Vector3D instance. |
| Vector3D_Dot | Calculates the dot product between two vectors. |
| Vector3D_GetX | Returns the X component of a Vector3D struct. |
| Vector3D_GetY | Returns the Y component of a Vector3D struct. |
| Vector3D_GetZ | Returns the Z component of a Vector3D struct. |
| Vector3D_Length | Calculated the length of a vector. |
| Vector3D_Maximum | Find the maximum vector from two source vectors. |
| Vector3D_Minimum | Find the minimum vector from two source vectors. |
| Vector3D_Multiply | Multiply two source vectors and place result in a destination vector. |
| Vector3D_MultiplyByMatrix | Multiplies the vector by a matrix. |
| Vector3D_Negate | Negates the given vector. |
| Vector3D_Normalize | Normalize the given vector. |

| | |
|---|---|
| Vector3D_OrientateByMatrix | Multiplies the vector by the orientation of the matrix. |
| Vector3D_Print | Prints the vector. |
| Vector3D_Scale | Scale a vector. |
| Vector3D_Set | Initializes a Vector3D struct. |
| Vector3D_Subtract | Subtract two vectors and store the result in a destination vector. |
| Vector3D_Zero | Zeros the vector. |

## Vector3D_Add

Add Add a source vector to a destination vector.

### Synopsis

```
void Vector3D_Add(Vector3D *v, Vector3D *a);
```

### Description

Add a source vector to a destination vector.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Average

Calculated the average of two vectors.

## Synopsis

```
void Vector3D_Average(Vector3D *v, Vector3D *a, Vector3D *b);
```

## Description

Calculated the average of two vectors.

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure |
| *a | Pointer to first source Vector3D structure |
| *b | |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Compare

Compare two source vectors.

### Synopsis

```
bool Vector3D_Compare(Vector3D *v, Vector3D *a);
```

### Description

Compare two source vectors.

### Arguments

| | |
|---|---|
| *v | Pointer to the first source Vector3D structure. |
| *a | Pointer to the second source Vector3D structure. |

### Return Value

| | |
|---|---|
| bool | TRUE Both vectors are exactly equal. |
| | FALSE Both vectors are not equal. |

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_CompareFuzzy

Compare two source vectors with fuzzy edges.

## Synopsis

```
bool Vector3D_CompareFuzzy(Vector3D *v, Vector3D *a, float
fuzzy);
```

## Description

Compare two source vectors with fuzzy edges.

## Arguments

| | |
|---|---|
| *v | Pointer to the first source Vector3D structure. |
| *a | Pointer to the second source Vector3D structure. |
| fuzzy | The fuzzy tolerance level. |

## Return Value

| | |
|---|---|
| bool | TRUE Both vectors are exactly equal. |
| | FALSE Both vectors are not equal. |

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Construct

Creates an instance of Vector3D.

### Synopsis

```
Vector3D* Vector3D_Construct(void);
```

### Description

Creates an instance of Vector3D.

### Arguments

void

### Return Value

| | |
|---|---|
| Vector3D* | Pointer to the new Vector3D structure. |

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Copy

Copy a source vector to a destination vector.

## Synopsis

```
void Vector3D_Copy(Vector3D *v, Vector3D *a);
```

## Description

Copy a source vector to a destination vector.

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the source Vector3D structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Cross

Calculates the cross product between two vectors.

### Synopsis

```
void Vector3D_Multiply(Vector3D *v, Vector3D *a, Vector3D
*b);
```

### Description

Calculates the cross product between two vectors and places result in destination vector.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the first source Vector3D structure. |
| *b | Pointer to the second source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Destruct

Deletes a Vector3D instance.

### Synopsis

```
void Vector3D_Destruct( Vector3D *v );
```

### Description

Deletes a Vector3D instance.

### Arguments

| | |
|---|---|
| *v | Pointer to the Vector3D structure to be deleted. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Dot

Calculates the dot product between two vectors.

### Synopsis

```
float Vector3D_Dot(Vector3D *v, Vector3D *a);
```

### Description

Calculates the dot product between two vectors.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the source Vector3D structure. |

### Return Value

float dot product value

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_GetX

Returns the X component of a Vector3D struct.

## Synopsis

```
float Vector3D_GetX(Vector3D *v);
```

## Description

Returns the X component of a Vector3D struct.

## Arguments

| | |
|---|---|
| *v | Pointer to a Vector3D structure. |

## Return Value

float X value.

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_GetY

Returns the Y component of a Vector3D struct.

### Synopsis

```
float Vector3D_GetY(Vector3D *v);
```

### Description

Returns the Y component of a Vector3D struct.

### Arguments

| | |
|---|---|
| *v | Pointer to a Vector3D structure. |

### Return Value

float Y value.

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_GetZ

Returns the Z component of a Vector3D struct.

## Synopsis

```
float Vector3D_GetZ(Vector3D *v);
```

## Description

Returns the Z component of a Vector3D struct.

## Arguments

| | |
|---|---|
| *v | Pointer to a Vector3D structure. |

## Return Value

float Z value.

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Compare(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Length(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Length

Calculates the length of a vector.

### Synopsis

```
float Vector3D_Length(Vector3D *v);
```

### Description

Calculates the length of a vector.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |

### Return Value

float length value.

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Maximum

Find the maximum vector from two source vectors.

## Synopsis

```
void Vector3D_Maximum(Vector3D *v, Vector3D *a, Vector3D *b);
```

## Description

Find the minimum vector from two source vectors.

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the first source Vector3D structure. |
| *b | Pointer to the second source Vector3D structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Minimum

Find the minimum vector from two source vectors.

### Synopsis

```
void Vector3D_Minimum(Vector3D *v, Vector3D *a, Vector3D *b);
```

### Description

Find the minimum vector from two source vectors.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the first source Vector3D structure. |
| *b | Pointer to the second source Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Multiply

Multiply two source vectors and place result in a destination vector.

## Synopsis

```
void Vector3D_Multiply(Vector3D *v, Vector3D *a, Vector3D
*b);
```

## Description

Multiply two source vectors and place result in a destination vector.
[ v.x = a.x * b.x ] [ v.y = a.y * b.y ] [ v.z = a.z * b.z ]

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the first source Vector3D structure. |
| *b | Pointer to the second source Vector3D structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(),
Vector3D_MultiplyByMatrix(),Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_MultiplyByMatrix

`Multiplies the vector by a matrix..`

### Synopsis

```
void Vector3D_MultiplyByMatrix( Vector3D *v, Matrix *m );
```

### Description

Multiplies the vector by a matrix.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *m | Pointer to the Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(),
Vector3D_Negate(),Vector3D_Multiply(), Vector3D_Dot(),
Vector3D_Length(), Vector3D_Normalize(), Vector3D_Minimum(),
Vector3D_Maximum(), Vector3D_Compare(), Vector3D_Copy(),
Vector3D_Add(), Vector3D_Sub(), Vector3D_Scale(),
Vector3D_Multiply(), Vector3D_Cross(), Vector3D_Zero(),
Vector3D_Print()
```

## Vector3D_Negate

Negates the given vector.

### Synopsis

```
void Vector3D_Negate(Vector3D *v);
```

### Description

Negates the given vector.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Normalize

Normalize the given vector.

### Synopsis

```
float Vector3D_Normalize(Vector3D *v);
```

### Description

Normalize the given vector.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_OrientateByMatrix

Multiplies the vector by the orientation of the matrix.

### Synopsis

```
void Vector3D_OrientateByMatrix( Vector3D *v, Matrix *m );
```

### Description

Multiplies the vector by the orientation of the matrix.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *m | Pointer to the Matrix structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Construct()
```

## Vector3D_Print

Prints the vector.

### Synopsis

```
void Vector3D_Print(Vector3D *v );
```

### Description

Prints the vector to standard output.

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Scale

Scale a vector.

## Synopsis

```
void Vector3D_Scale( Vector3D *v, float x, float y, float z );
```

## Description

Scale a vector in x,y & z. [ v.x *= x; v.y *= y; v.z *= z ]

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| x | X scale value. |
| y | Y scale value. |
| z | Z scale value. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Set

Initializes a Vector3D struct.

## Synopsis

```
void Vector3D_Set(Vector3D *v, float x, float y, float z);
```

## Description

Initializes a Vector3D struct.

## Arguments

| | |
|---|---|
| *v | Pointer to a Vector3D structure. |
| x | The new X value for the Vector3D. |
| y | The new Y value for the Vector3D. |
| z | The new Z value for the Vector3D. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Vector3D_Subtract

Subtract two vectors and store the result in a destination vector.

## Synopsis

```
void Vector3D_Subtract( Vector3D *v, Vector3D *a, Vector3D
*b);
```

## Description

Subtract two vectors and store the result in a destination vector. [v =a -b ]

## Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |
| *a | Pointer to the first source Vector3D structure. |
| *b | Pointer to the second source Vector3D structure. |

## Return Value

void

## Implementation

Version 1.0

## Associated Files

```
<matrix.h>, mercury/lib
```

## See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

## Vector3D_Zero

Zeros the vector.

### Synopsis

```
void Vector3D_Multiply(Vector3D *v, Vector3D *a, Vector3D
*b);
```

### Description

Zeros the vector. [ v.x = 0; v.y = 0; v.z = 0 ]

### Arguments

| | |
|---|---|
| *v | Pointer to the destination Vector3D structure. |

### Return Value

void

### Implementation

Version 1.0

### Associated Files

```
<matrix.h>, mercury/lib
```

### See Also

```
Vector3D_Set(), Vector3D_Average(), Vector3D_Negate(),
Vector3D_Dot(), Vector3D_Length(), Vector3D_Normalize(),
Vector3D_Minimum(), Vector3D_Maximum(), Vector3D_Compare(),
Vector3D_Copy(), Vector3D_Add(), Vector3D_Sub(),
Vector3D_Scale(), Vector3D_Multiply(), Vector3D_Cross(),
Vector3D_Zero(), Vector3D_Print()
```

# Index

## Symbols



## Numerics



## A



## B

# C

## D



## E



## F

## G



## H



## I



## L

## M

## N



## O



## P

## Q



## R

## S



## T

## U



## V



## W



## Z